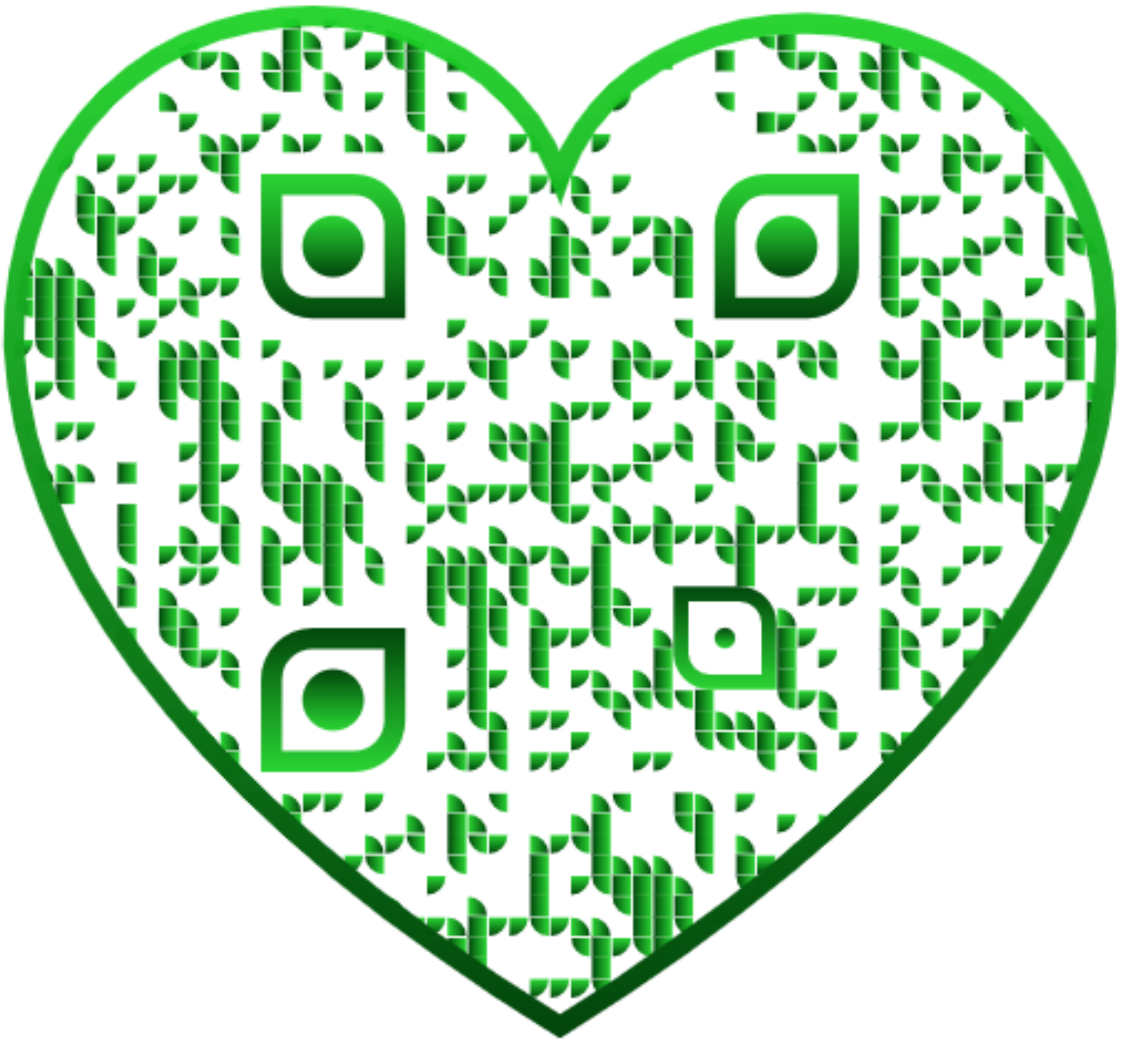
Maurice Chavelli

DÉCOUVREZ LE FRAMEWORK PHP

LARAVEL

OPENCLASSROOMS

EYROLLES



DÉCOUVREZ LE FRAMEWORK PHP

LARAVEL

Vous pratiquez PHP et vous savez créer des sites ? Vous avez l'impression de réécrire souvent les mêmes choses ? Vous vous posez des questions sur la meilleure façon de traiter une tâche particulière, comme créer des formulaires ou envoyer des e-mails ? Vous aimeriez disposer d'une boîte à outils toute prête pour tout le code laborieux ? Alors vous avez besoin d'un framework PHP et Laravel constitue actuellement ce qui se fait de mieux en la matière !

QU'ALLEZ-VOUS APPRENDRE ?

Les bases de Laravel

- Présentation générale
- Installation et organisation
- Routage et façades
- Les réponses
- Les contrôleurs
- Les entrées
- La validation
- Configuration et session
- L'injection de dépendances

Les bases de données

- Migrations et modèles
- Les ressources
- Ressources pour les utilisateurs et erreurs
- L'authentification
- Les relations 1:n et n:n
- Les commandes et les assistants
- Query Builder

Plus loin avec Laravel

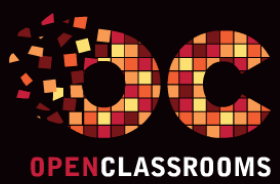
- Environnement et déploiement
- Des vues propres, avec ou sans le conteneur de dépendances
- La localisation
- Ajax
- Les tests unitaires
- Événements et autorisations

À PROPOS DE L'AUTEUR

Maurice Chavelli a commencé l'informatique sur un Sinclair ZX en 1981. Développeur .NET et administrateur réseau, il devient rapidement un acteur très présent sur le Web. Il ouvre en 2013 un blog sur le framework PHP Laravel. Créateur de plusieurs sites web, il découvre Bootstrap lors de sa sortie en 2011 et l'adopte rapidement en pressentant tout son potentiel !

L'ESPRIT D'OPENCLASSROOMS

Des cours ouverts, riches et vivants, conçus pour tous les niveaux et accessibles à tous gratuitement sur notre plate-forme d'e-éducation : www.openclassrooms.com. Vous y vivrez une véritable expérience communautaire de l'apprentissage, permettant à chacun d'apprendre avec le soutien et l'aide des autres étudiants sur les forums. Vous profiterez des cours disponibles partout, tout le temps : sur le Web, en PDF, en eBook, en vidéo…



www.editions-eyrolles.com

DÉCOUVREZ LE FRAMEWORK PHP

LARAVEL

Copyright © 2016 Eyrolles.

DANS LA MÊME COLLECTION

R. De Visscher. – **Découvrez le langage Swift.**
N°14397, 2016, 128 pages.

M. Lorant. – **Développez votre site web avec le framework Django.**
N°21626, 2015, 285 pages.

E. Lalitte. – **Apprenez le fonctionnement des réseaux TCP/IP.**
N°21623, 2015, 300 pages.

M. Nebra, M. Schaller. – **Programmez avec le langage C++.**
N°21622, 2015, 674 pages.

SUR LE MÊME THÈME

R. Goetter. – **CSS 3 Flexbox.**
N°14363, 2016, 152 pages.

W. McKinney. – **Analyse de données en Python.**
N°14109, 2015, 488 pages.

E. Biernat, M. Lutz. – **Data science : fondamentaux et études de cas.**
N°14243, 2015, 312 pages.

B. Philibert. – **Bootstrap 3 : le framework 100 % web design.**
N°14132, 2015, 318 pages.

C. Camin. – **Développer avec Symfony2.**
N°14131, 2015, 474 pages.

S. Pittion, B. Siebman. – **Applications mobiles avec Cordova et PhoneGap.**
N°14052, 2015, 184 pages.

C. Delannoy. – **Le guide complet du langage C.**
N°14012, 2014, 844 pages.

Retrouvez nos bundles (livres papier + e-book) et livres numériques sur
http://izibook.eyrolles.com

Copyright © 2016 Eyrolles.

Maurice Chavelli

DÉCOUVREZ LE FRAMEWORK PHP

LARAVEL

Copyright © 2016 Eyrolles.

OPENCLASSROOMS

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Copyright © 2016 Eyrolles.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans l'autorisation de l'Éditeur ou du Centre Français d'exploitation du droit de copie, 20, rue des Grands Augustins, 75006 Paris.

© Groupe Eyrolles, 2016. ISBN Eyrolles : 978-2-212-14398-0
© OpenClassrooms, 2016

Avant-propos

Vous pratiquez PHP et vous savez créer des sites ? Vous avez l'impression de réécrire souvent les mêmes choses ? Vous vous posez des questions sur la meilleure façon de traiter une tâche particulière comme créer des formulaires ou envoyer des courriels ? Vous aimeriez disposer d'une boîte à outils toute prête pour tout le code laborieux ? Vous devez choisir une solution logicielle pour développer un site dynamique ?

Alors vous avez besoin d'un *framework* PHP. Et Laravel constitue actuellement ce qui se fait de mieux en la matière.

Laravel colle aux plus récentes avancées de PHP et surtout à son approche objet. Découvrir ce framework et plonger dans son code c'est prendre un cours de programmation et d'esthétique. C'est aussi disposer d'une boîte à outils simple et performante pour construire des applications web sans se soucier de l'intendance habituelle.

Laravel est un framework encore jeune qui connaît un succès très rapide, surtout aux États-Unis où il est devenu le plus populaire. La documentation existant actuellement étant essentiellement en langue anglaise, le présent ouvrage a pour objectif de présenter ce framework très prometteur au public français. Il s'adresse principalement aux développeurs et chefs de projets.

Cet ouvrage a été conçu en trois parties progressives qu'il convient de lire dans l'ordre.

- La première partie, particulièrement détaillée, est destinée à vous habituer au framework et à sa philosophie. Elle vous présente les notions essentielles.
- La deuxième partie est axée sur les bases de données, qui constituent la clé des applications dynamiques. Les autres notions seront évidemment développées et complétées au cours de cette partie qui vous demandera plus d'attention et d'expérimentation.
- La troisième partie consolide vos connaissances et détaille quelques spécificités comme la manière d'obtenir des vues épurées, l'utilisation d'Ajax et la localisation. Elle présente la façon de déployer une application Laravel selon le serveur de destination.

Copyright © 2016 Eyrolles.

Le code est plus fourni et moins détaillé ; il vous faudra l'analyser et le mettre en pratique pour mettre à profit cet apprentissage.

Cet ouvrage nécessite d'avoir des connaissances correctes en PHP, HTML, CSS et JavaScript. Si vous avez des lacunes dans un de ces domaines, vous pouvez facilement les combler avec les nombreux cours à disposition sur le site OpenClassrooms (*www.openclassrooms.com*) et avec les nombreux ouvrages sur le sujet des éditions Eyrolles (*www.editions-eyrolles.com*) !

Copyright © 2016 Eyrolles.

Table des matières

Première partie – Les bases de Laravel **1**

1 Présentation générale **3**

Un framework ? 3
Approche personnelle 3
(Re)découvrir PHP 4
Définition et intérêt du framework 4

Pourquoi Laravel ? 4
Constitution de Laravel 4
Le meilleur de PHP 5
Documentation 6

Définitions de MVC et POO 6
MVC 6
POO 7

En résumé 8

2 Installation et organisation **9**

Composer 9
Présentation 9
Installation 10
Fonctionnement 10

Créer une application Laravel 11
Prérequis 11
Installation avec Composer 12
Installation avec Laravel Installer 13
Autorisations 14
Serveur 14
URL propres 14

Copyright © 2016 Eyrolles.

L'organisation de Laravel 15
Dossier app 15
Autres dossiers 16
Fichiers à la racine 17
Accessibilité 17
Environnement et messages d'erreur 17
Le composant Html 19
En résumé 20

3 Routage et façades **21**
Les requêtes HTTP 21
Petit rappel 21
.htaccess et index.php 22
Le cycle de la requête 22
Plusieurs routes et paramètres de route 24
Erreur d'exécution et contrainte de route 26
Une route nommée 27
Les façades 27
En résumé 29

4 Les réponses **31**
Construire une réponse 31
Codes des réponses 31
Vues 32
La vue paramétrée 33
URL 33
Route 34
Vue 34
Simplifier la syntaxe 35
Template 36
Les redirections 38
En résumé 38

5 Les contrôleurs **39**
L'utilité des contrôleurs 39
Rôle 39
Constitution 40
Liaison avec les routes 41
Route nommée 42
L'utilisation d'un contrôleur 42
En résumé 43

6 Les entrées **45**
Scénario et routes 45

Copyright © 2016 Eyrolles.

Le middleware 46
Le formulaire 47
Le contrôleur 49
La protection CSRF 51
En résumé 52

7 La validation **53**
Scénario et routes 53
Les vues 55
Template 55
Vue de contact 55
Vue de confirmation 58
Vue du courriel pour l'administrateur 59
La requête de formulaire 59
Le contrôleur 62
Envoyer un courriel 63
En résumé 64

8 Configuration et session **65**
La configuration 65
Les sessions 67
La requête de formulaire 68
Les routes et le contrôleur 69
Les vues 71
En résumé 74

9 L'injection de dépendances **75**
Le problème et sa solution 75
Problème 75
Solution 76
La gestion 78
En résumé 81

Deuxième partie – Les bases de données **83**

10 Migrations et modèles **85**
Les migrations 85
Configuration de la base 85
Artisan 86
Installer la migration 86
Créer la migration 87
Utiliser la migration 88

Copyright © 2016 Eyrolles.

Eloquent, un ORM très performant 89
Validation 90
Routes 92
Contrôleur 92
Vues 93
Fonctionnement 94
L'organisation du code 96
Première version 96
Deuxième version 97
Troisième version 98
En résumé 99

11 Les ressources **101**
Les données 101
Une ressource 103
Création 103
Routes 105
Contrôleur 106
La validation 107
Création d'un utilisateur 107
Modification d'un utilisateur 108
En résumé 109

12 Ressources pour les utilisateurs et erreurs **111**
Le gestionnaire de données (repository) 111
getPaginate 113
getById 114
show 114
edit 114
update 115
store 116
destroy 116
Les vues 117
Template 117
Vue index 117
Vue show 119
Vue edit 120
Vue create 121
Une réflexion sur le code 124
Gestionnaire de base 124
Modèle 125
Contrôleur 126
Les erreurs 128
En résumé 132

13 L'authentification **133**
La commande Artisan 133

Copyright © 2016 Eyrolles.

Les tables 134
users 134
password_reset 135
Les middlewares 136
Authenticate 136
RedirectIfAuthenticated 137
Routes et contrôleurs 138
Routes 138
Contrôleurs 139
AuthController 139
PasswordController 141
Les vues 141
L'enregistrement d'un utilisateur 143
Validation 143
Contrôleur 144
Vue auth.register 146
Connexion et déconnexion 147
Connexion 147
Vue auth.login 152
Déconnexion 152
L'oubli du mot de passe 153
En résumé 159

14 La relation 1:n **161**

Les données 161
Migrations 161
Population 164
La relation 167
Les modèles 168
Contrôleur et routes 170
Contrôleur 170
Routes 171
Le gestionnaire des articles 172
Les middlewares 173
La validation 174
Le fonctionnement 175
Liste des articles 175
Ajout d'un article 176
Suppression d'un article 176
Les vues 177
Template 177
Vue pour afficher les articles 178
Vue pour créer un article 181
En résumé 182

Copyright © 2016 Eyrolles.

15 La relation n:n **183**

Les données . . . 183
Migrations . . . 184
Population . . . 187

Les modèles . . . 189

La validation . . . 190

La gestion . . . 192
Contrôleur et routes . . . 192
Repositories . . . 193

Le fonctionnement . . . 195
Liste des articles . . . 195
Nouvel article . . . 196
Suppression d'un article . . . 198
Recherche par mot-clé . . . 198

Les vues . . . 199
Template . . . 199
Liste . . . 200
Vue de création d'un article . . . 203

En résumé . . . 204

16 Les commandes et les assistants **205**

Améliorer une commande . . . 205

Laravel Schema Designer . . . 213
Création des tables . . . 213
Création des champs . . . 214
Création des relations . . . 216
Exportation des fichiers . . . 219

En résumé . . . 222

17 Query Builder **223**

Les données . . . 223
Migration . . . 223
Population . . . 225

Les sélections . . . 228
Liste de tous les éditeurs . . . 228
Tableau des valeurs d'une colonne . . . 229
Ligne particulière . . . 229
Colonne isolée . . . 230
Lignes distinctes . . . 230
Plusieurs conditions . . . 231
Encadrer des valeurs . . . 232
Prendre des valeurs dans un tableau . . . 233
Ordonner et grouper . . . 233

Les jointures . . . 234
Trouver les titres des livres pour un éditeur dont on connaît l'identifiant . . . 234
Trouver les livres d'un auteur dont on connaît le nom . . . 235

Copyright © 2016 Eyrolles.

Trouver les auteurs pour un éditeur dont on connaît l'identifiant 236
Attention aux requêtes imbriquées 237
En résumé 238

Troisième partie – Plus loin avec Laravel **239**

18 Environnement et déploiement **241**
L'environnement 241
Le déploiement 243
Besoins de Laravel 243
Solution royale 244
Solution intermédiaire 245
Solution laborieuse 246
Mode maintenance 246
En résumé 247

19 Des vues propres **249**
Les macros 249
Problème 249
Définition 250
Fournisseur de services 252
Les templates 255
En résumé 257

20 Des vues propres avec le conteneur de dépendances **259**
La nouvelle vue 259
L'organisation du code 260
Les constructeurs 261
FormBuilder et HtmlBuilder 261
PanelBuilder 262
Fournisseur de services et façade 264
En résumé 267

21 La localisation **269**
Le principe 269
Façade Lang et classe Translator 269
Fichiers de langue 270
Fonctionnement 271
Le middleware 272
Les dates 275
Route et contrôleur 276
La réalisation de la localisation 278
Les vues 279

Copyright © 2016 Eyrolles.

Les noms des contrôles. 282
En résumé 283

22 Ajax **285**

Les vues 285
Template 285
Vue login 288
JavaScript 292
Le traitement 293
Contrôleur 293
Middleware 295
En résumé 296

23 Les tests unitaires **297**

L'intendance des tests. 298
PHPUnit 298
Intendance de Laravel 298
Environnement de test 300
Construire un test en trois étapes. 301
Assertions et appel de routes 302
Assertions 302
Appel de routes et test de réponse 302
Les vues et les contrôleurs 303
Vues 303
Contrôleurs 304
Isoler les tests 305
Simuler une classe 306
Tester une application. 308
En résumé 309

24 Événements et autorisations **311**

Les événements 311
Événements du framework 312
Fournisseur 313
Créer un observateur 317
Créer un événement 318
Les autorisations 319
Sécurité 319
Différentes autorisations 321
En résumé 324

Index **325**

Copyright © 2016 Eyrolles.

Première partie

Les bases de Laravel

Dans cette première partie, je vous propose de découvrir les bases de Laravel : où le trouver, comment l'installer, quelle configuration est nécessaire pour le faire fonctionner… Je vous présente aussi son organisation et sa philosophie : comment les requêtes sont aiguillées par des routes et dirigées vers des contrôleurs pour être traitées, et comment retourner ensuite une réponse appropriée.

Par ailleurs, vu qu'il est rare qu'une application web n'utilise pas de formulaire, je vous montre la manière d'en créer un, de valider les entrées saisies et de traiter les informations envoyées. La configuration de Laravel étant aussi une application avec ses propres nécessités et contraintes, nous verrons comment elle fonctionne.

Le protocole HTTP est performant mais il permet seulement des requêtes ponctuelles sans persistance. Or, lorsqu'on crée un site, on souhaite permettre aux utilisateurs de se connecter et d'avoir un environnement et des informations personnalisés. Il est donc nécessaire de gérer des sessions, et Laravel est bien équipé pour le faire.

Pour terminer cette partie, la notion d'injection de dépendance, qui est intensivement utilisée dans Laravel, sera présentée avec un exemple pratique d'application.

Copyright © 2016 Eyrolles.

Copyright © 2016 Eyro les.

1 Présentation générale

Dans ce premier chapitre, je vais évoquer PHP, son historique et sa situation actuelle. J'expliquerai aussi l'intérêt d'utiliser un framework pour ce langage et surtout pourquoi j'ai choisi Laravel. J'évoquerai enfin le patron MVC et la programmation orientée objet (POO).

Un framework ?

Approche personnelle

PHP est un langage populaire et accessible. Il est facile à installer et présent chez tous les hébergeurs. C'est un langage riche et plutôt facile à aborder, surtout pour quelqu'un qui a déjà des bases en programmation. On peut réaliser rapidement une application web fonctionnelle grâce à lui. Toutefois, le revers de cette simplicité est que, bien souvent, le code créé est confus, complexe et sans aucune cohérence. Il faut reconnaître que PHP n'encourage pas à organiser son code et rien n'oblige à le faire.

Lorsqu'on crée des applications PHP, on finit par avoir des codes personnels réutilisables pour les fonctionnalités récurrentes, par exemple pour gérer des pages de façon dynamique. Une fois qu'on a créé une fonction ou une classe pour réaliser une tâche il est naturel d'aller la chercher lorsque la même situation se présente. Puisque c'est une bibliothèque personnelle et qu'on est seul maître à bord, il faut évidemment la mettre à jour lorsque c'est nécessaire et c'est parfois fastidieux.

En général, on a aussi une hiérarchie de dossiers à laquelle on est habitué et on la reproduit quand on commence le développement d'une nouvelle application. On se rend compte parfois que cette habitude a des effets pervers, parce que la hiérarchie qu'on met ainsi en place de façon systématique n'est pas forcément la plus adaptée.

Copyright © 2016 Eyrolles.

En résumé, l'approche personnelle est plutôt du bricolage à la hauteur de ses compétences et de sa disponibilité.

(Re)découvrir PHP

Lorsque j'ai découvert PHP à la fin des années 1990, il en était à la version 3. C'était essentiellement un langage de script, en général mélangé au HTML, qui permettait de réaliser du *templating*, des accès aux données et du traitement. La version 4 en 2000 a apporté plus de stabilité et une ébauche de l'approche objet. Cependant, il a fallu attendre la version 5 en 2004 pour disposer d'un langage de programmation à la hauteur du standard existant pour les autres langages.

Cette évolution incite à perdre les mauvaises habitudes si on en avait. Un site comme *http://www.phptherightway.com* offre de bonnes pistes pour mettre en place de bonnes pratiques. Donc, si vous êtes un bidouilleur de code PHP, je vous conseille cette saine lecture qui devrait vous offrir un nouvel éclairage sur ce langage et, surtout, vous autoriser à vous lancer de façon correcte dans le code de Laravel.

Définition et intérêt du framework

D'après Wikipédia, un framework informatique est un « ensemble cohérent de composants logiciels structurels, qui sert à créer les fondations ainsi que les grandes lignes de tout ou d'une partie d'un logiciel » ; autrement dit, c'est une base cohérente avec des briques toutes prêtes à disposition. Il existe des frameworks pour tous les langages de programmation et en particulier pour PHP. En faire la liste serait laborieux tant il en existe !

L'utilité d'un framework est d'éviter de passer du temps à développer ce qui a déjà été fait par d'autres, souvent plus compétents, et qui a en plus été utilisé et validé par de nombreux utilisateurs. On peut imaginer un framework comme un ensemble d'outils à disposition. Par exemple, si je dois programmer du routage pour mon site, je prends un composant déjà tout prêt et qui a fait ses preuves et je l'utilise : gain de temps, fiabilité, mise à jour si nécessaire…

Il serait vraiment dommage de se passer d'un framework alors que le fait d'en utiliser un présente pratiquement uniquement des avantages.

Pourquoi Laravel ?

Constitution de Laravel

Laravel, créé par Taylor Otwel, initie une nouvelle façon de concevoir un framework en utilisant ce qui existe de mieux pour chaque fonctionnalité. Par exemple, toute application web a besoin d'un système qui gère les requêtes HTTP. Plutôt que de réinventer quelque chose, le concepteur de Laravel a tout simplement utilisé celui de **Symfony** en l'étendant pour créer un système de routage efficace. De la même manière, l'envoi

Copyright © 2016 Eyrolles.

des courriels se fait avec la bibliothèque *SwiftMailer*. En quelque sorte, Otwel a fait son marché parmi toutes les bibliothèques disponibles. Nous verrons dans cet ouvrage comment cela est réalisé. Néanmoins, Laravel n'est pas seulement le regroupement de bibliothèques existantes ; c'est aussi un ensemble de nombreux composants originaux et surtout une orchestration de tout cela.

Vous allez trouver dans Laravel :

- un système de routage perfectionné (*RESTFul* et ressources) ;
- un créateur de requêtes SQL et un ORM performants ;
- un moteur de templates efficace ;
- un système d'authentification pour les connexions ;
- un système de validation ;
- un système de pagination ;
- un système de migration pour les bases de données ;
- un système d'envoi de courriels ;
- un système de cache ;
- un système d'événements ;
- un système d'autorisations ;
- une gestion des sessions...
- et bien d'autres choses encore que nous allons découvrir ensemble.

Il est probable que certains éléments de cette liste ne vous évoquent pas grand-chose, mais ce n'est pas important pour le moment ; tout cela deviendra plus clair au fil des chapitres.

Le meilleur de PHP

Plonger dans le code de Laravel, c'est recevoir un cours de programmation tant le style est clair et élégant et le code merveilleusement organisé. La version actuelle de Laravel est la 5.2 et nécessite au minimum la version 5.5.9 de PHP. Pour aborder de façon efficace ce framework, il est souhaitable de se familiariser avec les notions suivantes.

- **Les espaces de noms :** c'est une façon de bien ranger le code pour éviter des conflits de nommage. Laravel utilise cette possibilité de façon intensive. Tous les composants sont rangés dans des espaces de noms distincts, de même que l'application créée.
- **Les fonctions anonymes :** ce sont des fonctions sans nom (souvent appelées *closures*) qui améliorent le code. Les utilisateurs de JavaScript y sont habitués, ceux de PHP un peu moins parce qu'elles y sont plus récentes. Laravel les utilise aussi de façon systématique.
- **Les méthodes magiques :** ce sont des méthodes qui n'ont pas été explicitement décrites dans une classe, mais qui peuvent être appelées et résolues.
- **Les interfaces :** une interface est un contrat de constitution des classes. En programmation objet, c'est le sommet de la hiérarchie. Tous les composants de Laravel

Copyright © 2016 Eyrolles.

sont fondés sur des interfaces. La version 5 a même vu apparaître un lot de contrats pour étendre de façon sereine le framework.

- **Les traits :** c'est une façon d'ajouter des propriétés et méthodes à une classe sans passer par l'héritage, ce qui aide à passer outre certaines limitations de l'héritage simple proposé par défaut par PHP.

Un framework n'est pas fait pour remplacer la connaissance d'un langage, mais pour assister celui (ou celle) qui connaît déjà bien ce langage. Si vous avez des lacunes, il vaut mieux les combler pour profiter pleinement de Laravel.

Documentation

Quand on s'intéresse à un framework, il ne suffit pas qu'il soit riche et performant ; il faut aussi que la documentation soit à la hauteur. C'est le cas pour Laravel. Vous trouverez la documentation sur le site officiel *https://laravel.com/*, mais il existe de plus en plus d'autres sources, dont voici les principales :

- *https://laravel.fr/* : site d'entraide francophone avec un forum actif ;
- *http://laravel.io/* : le forum officiel ;
- *http://laravel.sillo.org/* : mon blog créé début 2013 et toujours actif, qui constitue une initiation progressive complémentaire du présent ouvrage ;
- *http://cheats.jesse-obrien.ca/* : une page bien pratique qui résume toutes les commandes ;
- *http://www.laravel-tricks.com/* : un autre site d'astuces ;
- *http://packalyst.com/* : le rassemblement de tous les *packages* pour ajouter des fonctionnalités à Laravel ;
- *https://laracasts.com/* : de nombreux tutoriels vidéo en anglais, dont un certain nombre en accès gratuit, notamment une série complète pour Laravel 5 : *https://laracasts.com/series/laravel-5-fundamentals*.

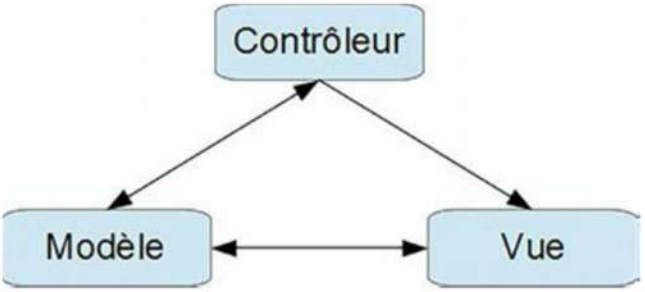
Il existe aussi de bon livres, mais ils sont tous en anglais.

Définitions de MVC et POO

MVC

On peut difficilement parler d'un framework sans évoquer le patron *Modèle-Vue-Contrôleur*. Pour certains, il s'agit de la clé de voûte de toute application rigoureuse ; pour d'autres, c'est une contrainte qui empêche d'organiser judicieusement son code. De quoi s'agit-il ? Voici un petit schéma pour y voir clair.

Copyright © 2016 Eyrolles.



Le modèle MVC

C'est un modèle d'organisation du code.

- Le *modèle* est chargé de gérer les données.
- La *vue* est chargée de la mise en forme pour l'utilisateur.
- Le *contrôleur* est chargé de gérer l'ensemble.

En général, on résume en disant que le modèle gère la base de données, la vue produit les pages HTML et le contrôleur fait tout le reste. Plus précisément, détaillons cette organisation dans Laravel.

- Le modèle correspond à une table d'une base de données. C'est une classe qui étend la classe `Model`, qui gère simplement et efficacement les manipulations de données et l'établissement automatisé de relations entre tables.
- Le contrôleur se décline en deux catégories : contrôleur classique et contrôleur de ressource (je détaillerai évidemment tout cela dans cet ouvrage).
- La vue est soit un simple fichier avec du code HTML, soit un fichier utilisant le système de templates `Blade` de Laravel.

Laravel propose ce modèle mais ne l'impose pas. Nous verrons d'ailleurs qu'il est parfois judicieux de s'en éloigner, parce qu'il y a plusieurs choses qu'on n'arrive pas à caser dans ce modèle. Par exemple, si je dois envoyer des emails, où vais-je placer mon code ? En général, ce qui se produit est l'inflation des contrôleurs auxquels on demande des rôles pour lesquels ils ne sont pas faits.

POO

Laravel est fondamentalement orienté objet. La POO est un *design pattern* qui s'éloigne radicalement de la programmation procédurale. Avec la POO, tout le code est placé dans des classes qui découlent d'interfaces établissant des contrats de fonctionnement. Avec la POO, on manipule des objets.

Avec la POO, la responsabilité du fonctionnement est répartie dans des classes alors que dans l'approche procédurale tout est mélangé. Le fait de répartir la responsabilité évite la duplication du code qui est le lot presque forcé de la programmation procédurale. Laravel pousse au maximum cette répartition en utilisant l'injection de dépendances.

L'utilisation de classes bien identifiées dont chacune a un rôle précis, le pilotage par des interfaces claires et l'injection de dépendances, tout cela crée un code élégant, efficace, lisible, facile à maintenir et à tester. C'est ce que Laravel propose. Alors, vous

Copyright © 2016 Eyrolles.

pouvez évidemment greffer là-dessus votre code approximatif, mais vous pouvez aussi vous inspirer des sources du framework pour améliorer votre style de programmation.

L'injection de dépendances est destinée à éviter de rendre les classes dépendantes et à privilégier une liaison dynamique plutôt que statique. Le résultat est un code plus lisible, plus facile à maintenir et à tester. Nous verrons ce mécanisme à l'œuvre dans Laravel.

En résumé

- Un framework fait gagner du temps et donne l'assurance de disposer de composants bien codés et fiables.
- Laravel est un framework novateur, complet, qui utilise les possibilités les plus récentes de PHP et qui est impeccablement codé et organisé.
- La documentation de Laravel est complète, précise et de plus en plus de tutoriels et exemples apparaissent sur la toile.
- Laravel adopte le patron MVC, mais ne l'impose pas. Il est totalement orienté objet.

Copyright © 2016 Eyrolles.

2 Installation et organisation

Dans ce chapitre, nous allons faire connaissance avec le gestionnaire de dépendances Composer, présenter la création d'une application Laravel et expliquer comment le code est organisé dans cette application.

Avertissement

Pour utiliser Laravel et suivre ce chapitre et l'ensemble de l'ouvrage, vous aurez besoin d'un serveur équipé de PHP avec au minimum la version 5.5.9 et aussi de MySQL. Il existe plusieurs applications « tout-en-un » faciles à installer : `wampserver` (*http://www.wampserver.com/*), `xampp` (*https://www.apachefriends.org/fr/index.html*), `easyphp` (*http://www.easyphp.org/*)... Personnellement, j'utilise `wamp`, qui répond sans problème à toutes mes attentes et permet de basculer entre les versions de PHP et de MySQL en un simple clic.
Une solution toute prête, `Homestead` (*https://laravel.com/docs/5.0/homestead*), est facile à mettre en œuvre sous Linux, mais beaucoup moins conviviale avec Windows. Pour ce dernier, il existe une autre possibilité bien pensée : `Laragon` (*https://laragon.org/*).
Quelle que soit l'application que vous utilisez, vérifiez que vous avez la bonne version de PHP (minimum 5.5.9). En outre, les extensions `PDO`, `Tokenizer`, `OpenSSL` et `Mbstring` de PHP doivent être activées.

Composer

Présentation

Je vous ai dit que Laravel utilise des composants d'autres sources. Plutôt que de les incorporer directement, il utilise un gestionnaire de dépendances : Composer. D'ailleurs, les composants de Laravel sont aussi traités comme des dépendances. De quoi s'agit-il ?

Imaginez que vous créez une application PHP et que vous utilisez des composants issus de différentes sources : `Carbon` pour les dates, `Redis` pour les données… Une

Copyright © 2016 Eyrolles.

méthode laborieuse consiste à aller chercher tout cela de façon manuelle, et vous allez être confrontés à des difficultés :

- télécharger tous les composants dont vous avez besoin et les placer dans votre structure de dossiers ;
- traquer les éventuels conflits de nommage entre les bibliothèques ;
- mettre à jour manuellement les bibliothèques quand c'est nécessaire ;
- prévoir le code pour charger les classes à utiliser…

Tout cela est évidemment faisable, mais avouez qu'il serait bien agréable d'automatiser ces procédures. C'est justement ce que fait un gestionnaire de dépendances !

Installation

Laravel utilise Composer comme gestionnaire de dépendances. Il vous faut donc commencer par l'installer sur votre ordinateur. Selon votre système, la procédure est différente ; je vous renvoie donc au site *https://getcomposer.org/* pour obtenir tous les renseignements sur le sujet.

Pour Windows, il suffit de télécharger un installeur (*https://getcomposer.org/download/*), qui fait tout très proprement et renseigne aussi la variable d'environnement `PATH`, ce qui rend Composer utilisable depuis n'importe quel emplacement. En revanche, l'installeur vous demandera où se trouve `php.exe` et vous devrez répondre, car Composer est un fichier PHP et a besoin d'être exécuté.

Pour les autres systèmes, en particulier Linux, le plus simple est d'utiliser `curl`. Il suffit de suivre les instructions détaillées sur le site.

Pour aller plus loin avec Composer, vous pouvez lire l'article suivant : *http://laravel.sillo.org/jouer-avec-composer/*.

Fonctionnement

Pour comprendre le fonctionnement de Composer, il faut connaître le format JSON (*JavaScript Object Notation*). Un fichier JSON a pour but de contenir des informations de type étiquette-valeur. Regardez cet exemple élémentaire :

```
{
  "nom": "Durand",
  "prénom": "Jean"
}
```

Les étiquettes sont `"nom"` et `"prénom"` ; les valeurs correspondantes sont `"Durand"` et `"Jean"`. Les valeurs peuvent être aussi des tableaux ou des objets. Regardez ce second exemple :

Copyright © 2016 Eyrolles.

```
{
  "identité1" : {
    "nom": "Durand",
    "prénom": "Jean"
  },
  "identité2" : {
    "nom": "Dupont",
    "prénom": "Albert"
  }
}
```

Composer a besoin d'un fichier `composer.json`, qui contient les instructions nécessaires : les dépendances, les classes à charger automatiquement… Voici un extrait de ce fichier pour Laravel :

```
{
  "name": "laravel/laravel",
  "description": "The Laravel Framework.",
  "keywords": ["framework", "laravel"],
  "license": "MIT",
  "type": "project",
  "require": {
    "php": ">=5.5.9",
    "laravel/framework": "5.2.*"
  },
  ...
}
```

Créer une application Laravel

Prérequis

Composer fonctionne en ligne de commande. Vous avez donc besoin de la console (nommée Terminal ou Konsole sur OS X et Linux). Les utilisateurs de Linux sont très certainement habitués à l'utilisation de la console, mais il n'en est généralement pas de même pour les adeptes de Windows. Pour trouver la console sur ce système, il faut chercher l'invite de commande.

Trouver la console dans Windows

Copyright © 2016 Eyrolles.

Cet ouvrage a été créé avec la version 5.2.* de Laravel. Lorsque vous créez une nouvelle application, que ce soit avec `composer create-project` ou avec l'installeur, vous obtenez la dernière version stable. Sur *OpenClassrooms.com*, je m'efforcerai de garder ce cours en phase avec l'évolution de Laravel, mais il y aura toujours un délai entre la sortie d'une nouvelle version et cet ouvrage. Si vous rencontrez des différences de fonctionnement avec les exemples utilisés, vous pouvez toujours, en attendant la mise à niveau de l'ouvrage installer la version précédente de Laravel. Il suffit d'utiliser la commande `create-project` en spécifiant la version comme troisième argument (voir la documentation complète sur *https://getcomposer.org/doc/03-cli.md#create-project*).

Installation avec Composer

Il y a plusieurs façons de créer une application Laravel. Celle qui me semble la plus simple consiste à utiliser la commande `create-project` de Composer. Par exemple, si je veux créer une application dans un dossier `laravel5` à la racine de mon serveur, voici la syntaxe à utiliser :

```
composer create-project --prefer-dist laravel/laravel laravel5
```

L'installation démarre et je n'ai plus qu'à attendre quelques minutes pour que Composer fasse son travail jusqu'au bout. Une liste de téléchargements s'affiche, pour finalement se retrouver avec l'architecture suivante.

Architecture des dossiers de Laravel

Copyright © 2016 Eyrolles.

On peut vérifier que tout fonctionne bien avec l'URL *http://localhost/laravel5/public*. Normalement, on doit obtenir cette page très épurée :

Page d'accueil de Laravel

Sous Windows avec `Wamp`, il est possible d'avoir un souci pour afficher la page d'accueil. Si c'est votre cas, il y a trois solutions.

- N'utilisez pas `Wamp` mais par exemple `Laragon` (*http://laragon.org/*).
- Créez un hôte virtuel.
- Suivez ce qui est préconisé sur la page *http://stackoverflow.com/questions/15607132/laravel-route-not-working-with-wamp*.

Pour les mises à jour ultérieures, il suffit d'utiliser encore Composer, mais avec la commande `update` :

```
composer update
```

Installation avec Laravel Installer

Une autre solution consiste à utiliser l'installeur de Laravel. Il faut commencer par installer globalement l'installeur avec Composer :

```
composer global require "laravel/installer"
```

Il faut ensuite informer la variable d'environnement `path` de l'emplacement du dossier `.../composer/vendor/bin`.

Pour créer une application, il suffit de taper :

```
laravel new monAppli
```

Laravel sera alors installé dans le dossier `monAppli`.

Si vous installez Laravel en téléchargeant directement les fichiers sur Github et en utilisant la commande `composer install`, il vous faut effectuer deux actions complémentaires. Dans ce cas en effet, il ne sera pas automatiquement créé de clé de sécurité et vous allez tomber sur une erreur au lancement. Il faut donc la créer avec la commande `php artisan key:generate`. De plus, vous aurez à la racine le fichier `.env.example` que vous devrez renommer en `.env` pour que la configuration fonctionne.

Copyright © 2016 Eyrolles.

Autorisations

Au niveau des dossiers de Laravel, le seul qui ait besoin de droits d'écriture par le serveur est `storage`.

Serveur

Pour fonctionner correctement, Laravel a besoin de PHP :

- version ≥ 5.5.9 ;
- extension `PDO` ;
- extension `Mbstring` ;
- extension `OpenSSL` ;
- extension `Tokenizer`.

URL propres

Pour un serveur Apache, il est prévu dans le dossier `public` un fichier `.htaccess` avec ce code :

```
<IfModule mod_rewrite.c>
  <IfModule mod_negotiation.c>
    Options -MultiViews
  </IfModule>

  RewriteEngine On

  # Redirect Trailing Slashes If Not A Folder...
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteRule ^(.*)/$ /$1 [L,R=301]

  # Handle Front Controller...
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteRule ^ index.php [L]

  # Handle Authorization Header
  RewriteCond %{HTTP:Authorization} .
  RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
</IfModule>
```

Le but est d'éviter d'avoir `index.php` dans l'URL. Cependant, pour que cela fonctionne, il faut activer le module `mod_rewrite`.

Une autre façon d'obtenir un Laravel sur mesure est d'utiliser mon outil en ligne *http://laravel-designer.sillo.org/*, décrit à l'adresse *http://laravel.sillo.org/laravel-designer/*.

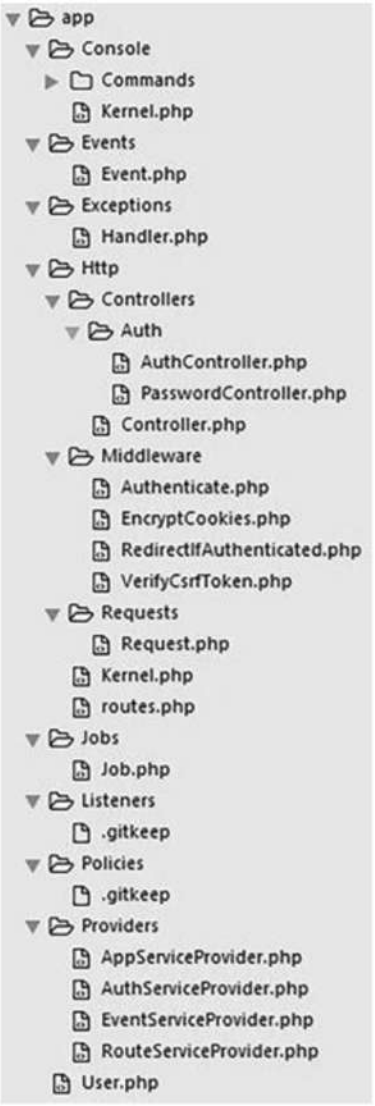
Copyright © 2016 Eyrolles.

L'organisation de Laravel

Maintenant qu'on a un Laravel tout neuf et qui fonctionne, voyons un peu ce qu'il contient.

Dossier app

Ce dossier contient les éléments essentiels de l'application.



Dossier App

- `Console/Commands` : toutes les commandes en mode console (la commande `Inspire` sert d'exemple).
- `Jobs` : commandes concernant les tâches que doit effectuer l'application (nouveauté de la version 5 que je n'aborderai pas dans cet ouvrage).

Copyright © 2016 Eyrolles.

- `Events` et `Listeners` : événements et écouteurs nécessaires pour l'application.
- `Http` : tout ce qui concerne la communication : contrôleurs, routes, *middlewares* (il y a quatre *middlewares* de base) et requêtes.
- `Providers` : tous les fournisseurs de services (quatre au départ), qui servent à initialiser les composants.
- `Policies` : une évolution récente qui facilite la gestion des droits d'accès.
- `Exceptions` : tout ce qui concerne les erreurs d'exécution.

On trouve également le fichier `User.php`, qui est un modèle d'utilisateur pour la base de données.

Évidemment, tout cela doit vous paraître assez nébuleux pour le moment, mais nous verrons en détail la plupart de ces sections au fil de l'ouvrage.

Autres dossiers

Autres dossiers

Voici une description du contenu des autres dossiers :

- `bootstrap` : scripts d'initialisation de Laravel pour le démarrage de l'application, le chargement automatique des classes et la fixation de l'environnement et des chemins ;
- `public` : tout ce qui doit apparaître dans le dossier public du site (images, CSS, scripts, etc.) ;
- `vendor` : tous les composants de Laravel et de ses dépendances ;
- `config` : toutes les configurations (application, authentification, cache, base de données, espaces de noms, emails, systèmes de fichiers, session, etc.) ;
- `database` : migrations et populations ;
- `resources` : vues, fichiers de langage et *assets* (par exemple les fichiers LESS ou Sass) ;
- `storage` : données temporaires de l'application (vues compilées, caches, clés de session, etc.) ;
- `tests` : fichiers de tests unitaires.

Copyright © 2016 Eyrolles.

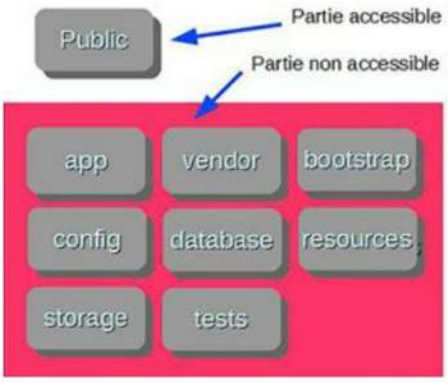
Fichiers à la racine

Il y a un certain nombre de fichiers à la racine. En voici les principaux :

- `artisan` : outil en ligne de Laravel pour des tâches de gestion ;
- `composer.json` : fichier de référence de Composer ;
- `phpunit.xml` : fichier de configuration de `phpunit` (pour les tests unitaires) ;
- `.env` : fichier pour spécifier l'environnement d'exécution.

Accessibilité

Pour des raisons de sécurité sur le serveur, seul le dossier `public` doit être accessible.



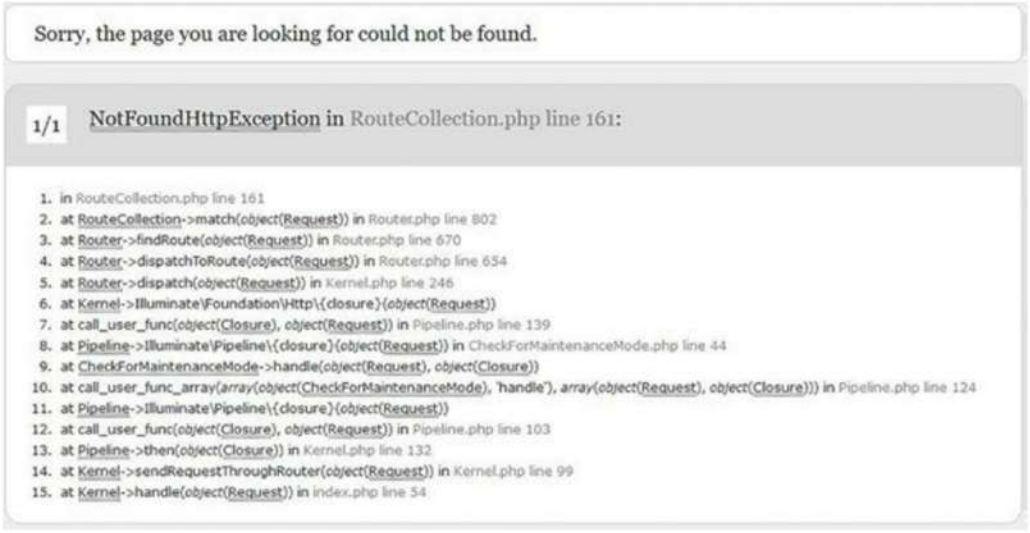
Le dossier public est le seul accessible

Cette configuration n'est pas toujours possible sur un serveur mutualisé. Il faut alors modifier un peu Laravel pour que cela fonctionne ; j'en parlerai dans le chapitre sur le déploiement.

Environnement et messages d'erreur

Par défaut, lorsque vous installez Laravel, celui-ci est en mode *debug*. Concernant l'affichage des erreurs, si vous entrez une URL qui n'est pas prévue, vous obtenez quelque chose comme ceci.

Copyright © 2016 Eyrolles.



Un message d'erreur en mode debug

Pendant la phase de développement, on a besoin d'obtenir des messages explicites pour traquer les erreurs qu'on commettra inévitablement. En mode *production* au contraire, il faudra cacher tous ces détails. Ouvrez le fichier `config/app.php` et trouvez la ligne suivante :

```
'debug' => env('APP_DEBUG', false),
```

Autrement dit, on cherchera la valeur dans l'environnement. Regardez à la racine des dossiers, vous y trouvez un fichier `.env`.

Le fichier de l'environnement

En voici le contenu :

```
APP_ENV=local
APP_DEBUG=true
APP_KEY=base64:/qnv1yuwcCDFJuki91gc7LBtIzRkJgFxusIX2x1wwUM=
APP_URL=http://localhost

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=homestead
```

Copyright © 2016 Eyrolles.

```
DB_USERNAME=homestead
DB_PASSWORD=secret

CACHE_DRIVER=file
SESSION_DRIVER=file
...
```

Vous remarquez que, dans ce fichier, la variable `APP_DEBUG` a la valeur `true`. On la conserve ainsi pour être en mode *debug*, avec affichage de messages d'erreur détaillés. Pour passer en mode *production*, il faut indiquer `false` (ou supprimer la variable). Avec une URL non prévue, vous obtenez alors juste ce qui suit.

Un message d'erreur en mode production

Il ne faudra évidemment pas laisser la valeur `true` lors d'une mise en production ! On en reparlera lorsqu'on verra la gestion de l'environnement. Vous ne risquerez ainsi plus d'oublier de changer cette valeur parce que Laravel saura si vous êtes sur votre serveur de développement ou sur celui de production.

La valeur de `APP_KEY` qui sécurise les informations est automatiquement générée lors de l'installation avec `create-project`.

Le composant Html

Dans la version 4 de Laravel, le composant `Html` facilitait la création des formulaires et offrait un lot de fonctions dites *helpers* pour faciliter l'écriture du HTML. Dans la version 5, ce composant n'est pas chargé par défaut. Comme nous en aurons besoin, une fois que vous avez réussi à installer une application toute neuve de Laravel, modifiez comme suit le fichier `composer.json` :

```
"require": {
  "php": ">=5.5.9",
  "laravel/framework": "5.2.*",
  "laravelcollective/html": "5.2.*"
},
```

On demande ainsi à Composer de charger le composant `laravelcollective/html` (*https://laravelcollective.com/docs/5.2/html*). Lancez alors une mise à jour (attention de bien vous positionner dans le dossier racine de l'application) :

Copyright © 2016 Eyrolles.

```
composer update
```

Attendez la fin du chargement. Il faut ensuite modifier comme suit le fichier `config/app.php` (ajout des lignes contenant `Collective\Html`) :

```
<?php
  /*
   * Application Service Providers...
   */
  App\Providers\AppServiceProvider::class,
  App\Providers\AuthServiceProvider::class,
  App\Providers\EventServiceProvider::class,
  App\Providers\RouteServiceProvider::class,
  Collective\Html\HtmlServiceProvider::class,
  ...

  'View'=>Illuminate\Support\Facades\View::class,
  'Form'=>Collective\Html\FormFacade::class,
  'Html'=>Collective\Html\HtmlFacade::class,
```

Ainsi, vous allez disposer de ce composant bien utile !

Le composant utilisé est dérivé de `Illuminate/html`, qui ne sera plus suivi.

En résumé

- Pour son installation et sa mise à jour, Laravel utilise le gestionnaire de dépendances `Composer`.
- La création d'une application Laravel se fait à partir de la console avec une simple ligne de commande.
- Laravel est organisé en plusieurs dossiers.
- Le dossier `public` est le seul qui doive être accessible pour le client.
- L'environnement est fixé à l'aide du fichier `.env`.
- Le composant `Html` n'est pas prévu par défaut ; il faut le charger indépendamment.

Copyright © 2016 Eyrolles.

3 Routage et façades

Dans ce chapitre, nous allons nous intéresser au devenir d'une requête HTTP qui arrive dans notre application Laravel. Nous allons voir l'intérêt d'utiliser un fichier `.htaccess` pour simplifier les URL. Nous présenterons aussi le système de routage pour trier les requêtes.

Les requêtes HTTP

Petit rappel

Commençons par un petit rappel sur ce qu'est une requête HTTP.

Les requêtes HTTP

HTTP *(Hypertext Transfer Protocol)* est un protocole de communication entre un client et un serveur. Le client demande une page au serveur en envoyant une *requête* et le serveur retourne une *réponse*, en général une page HTML.

Copyright © 2016 Eyrolles.

La requête du client comporte un certain nombre d'informations, mais nous allons nous intéresser pour le moment seulement à deux d'entre elles :

- la *méthode* : `get`, `post`, `put`, `delete` ;
- l'*URL* : c'est l'adresse de la page demandée sur le serveur.

Notre application Laravel doit savoir interpréter ces informations et les utiliser de façon pertinente pour renvoyer ce que demande le client.

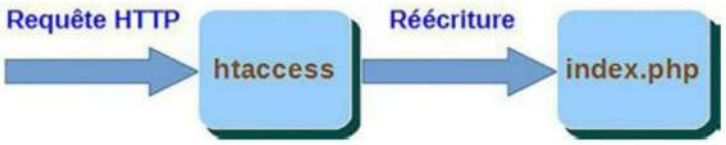
.htaccess et index.php

On veut que toutes les requêtes aboutissent obligatoirement sur le fichier `index.php` situé dans le dossier `public`. Pour y arriver, on peut utiliser une URL de ce genre :

```
http://monsite.fr/index.php/mapage
```

Cependant, ce n'est pas très esthétique avec cet `index.php` au milieu. Si vous avez un serveur Apache, lorsque la requête du client arrive sur le serveur où se trouve votre application Laravel, elle passe en premier par le fichier `.htaccess`, s'il existe, qui fixe des règles pour le serveur. Il y a justement un tel fichier dans le dossier `public` de Laravel, qui contient une règle de réécriture fournissant des URL simplifiées :

```
http://monsite.fr/mapage
```



La réécriture des URL

Pour que cela fonctionne, il faut que le serveur Apache ait le module `mod_rewrite` activé.

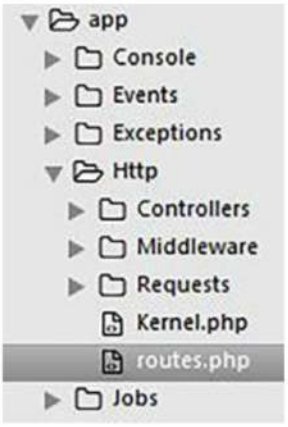
Si vous n'utilisez pas Apache mais Nginx, il faut utiliser la directive suivante :

```
location / {
    try_files $uri $uri/ /index.php?$query_string;
}
```

Le cycle de la requête

Lorsque la requête atteint le fichier `public/index.php`, l'application Laravel est créée et configurée, puis l'environnement est détecté. Nous reviendrons plus tard en détail sur ces étapes. Ensuite, le fichier `routes.php` est chargé.

Copyright © 2016 Eyrolles.



Les fichiers des routes

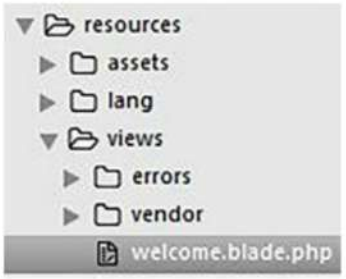
C'est avec ce fichier que la requête sera analysée et dirigée. Regardons ce qu'on y trouve au départ :

```
<?php
Route::get('/', function () {
  return view('welcome');
});
```

Comme Laravel est explicite, vous pouvez déjà deviner à quoi sert ce code :

- `Route` : on utilise le routeur ;
- `get` : on regarde si la requête a la méthode `get` ;
- `'/'` : on regarde si l'URL comporte uniquement le nom de domaine ;
- dans la fonction anonyme, on retourne (`return`) une vue (`view`) à partir du fichier `welcome`.

Ce fichier `welcome` se trouve bien rangé dans le dossier des vues.



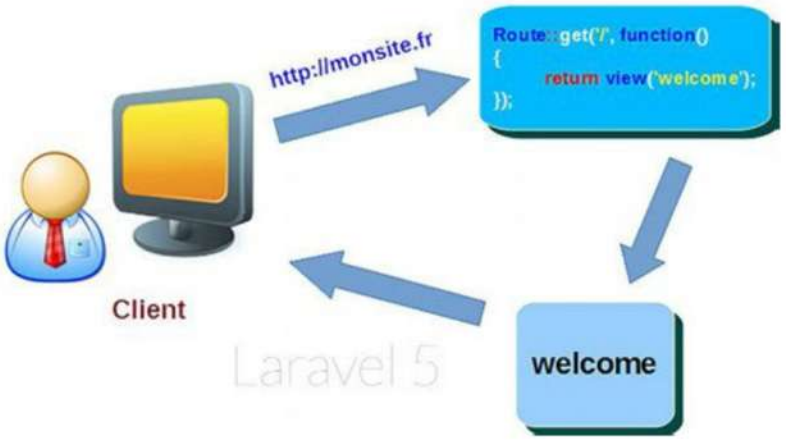
La vue welcome dans le dossier des vues

C'est ce fichier comportant du code HTML qui génère le texte d'accueil obtenu au démarrage initial de Laravel.

Copyright © 2016 Eyrolles.

Laravel propose plusieurs fonctions *helpers* qui simplifient la syntaxe, facilitant et accélérant le codage. Il y a par exemple `view` pour la classe `View` comme on l'a vu dans le code précédent.

Visualisons le cycle de la requête sur la figure suivante.



Le cycle de la requête

Sur votre serveur local, vous n'avez pas de nom de domaine et vous allez utiliser une URL de la forme `http://localhost/tuto/public` en admettant que vous ayez créé Laravel dans un dossier `www/tuto`. Vous pouvez aussi créer un hôte virtuel pour avoir une situation plus réaliste.

Plusieurs routes et paramètres de route

À l'installation, Laravel connaît une seule route, qui correspond à l'URL de base composée uniquement du nom de domaine. Voyons maintenant comment créer d'autres routes. Imaginons que nous ayons trois pages qui doivent être affichées avec les URL suivantes :

- *http://monsite.fr/1* ;
- *http://monsite.fr/2* ;
- *http://monsite.fr/3*.

J'ai fait apparaître en gras la partie spécifique de l'URL pour chaque page. Il est facile de réaliser cela avec le code suivant :

```
<?php
Route::get('1', function() {return 'Je suis la page 1 !';});
Route::get('2', function() {return 'Je suis la page 2 !';});
Route::get('3', function() {return 'Je suis la page 3 !';});
```

Copyright © 2016 Eyrolles.

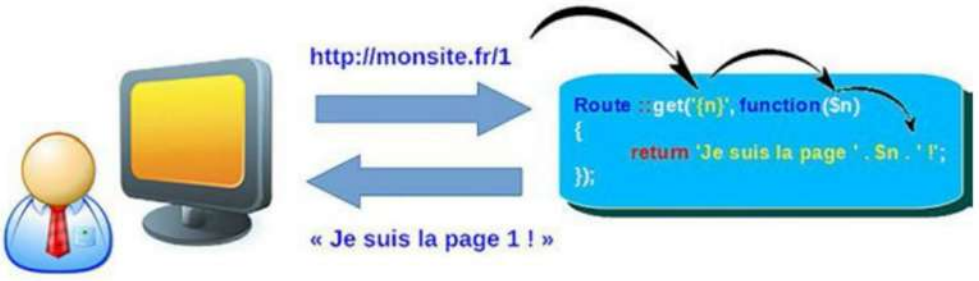
Cette fois, je n'ai pas créé de vue parce que ce qui nous intéresse est uniquement une mise en évidence du routage ; je retourne donc directement la réponse au client. Visualisons cela pour la page 1.

Demande de la page 1

On a besoin du caractère / uniquement dans la route de base.

On peut maintenant se poser une question : est-il vraiment indispensable de créer trois routes alors que la seule différence tient juste à une valeur qui change ? On peut utiliser un paramètre pour une route qui accepte des éléments variables en utilisant des accolades :

```
<?php
Route::get('{n}', function($n) {
  return 'Je suis la page ' . $n .' !';
});
```



Une route paramétrée

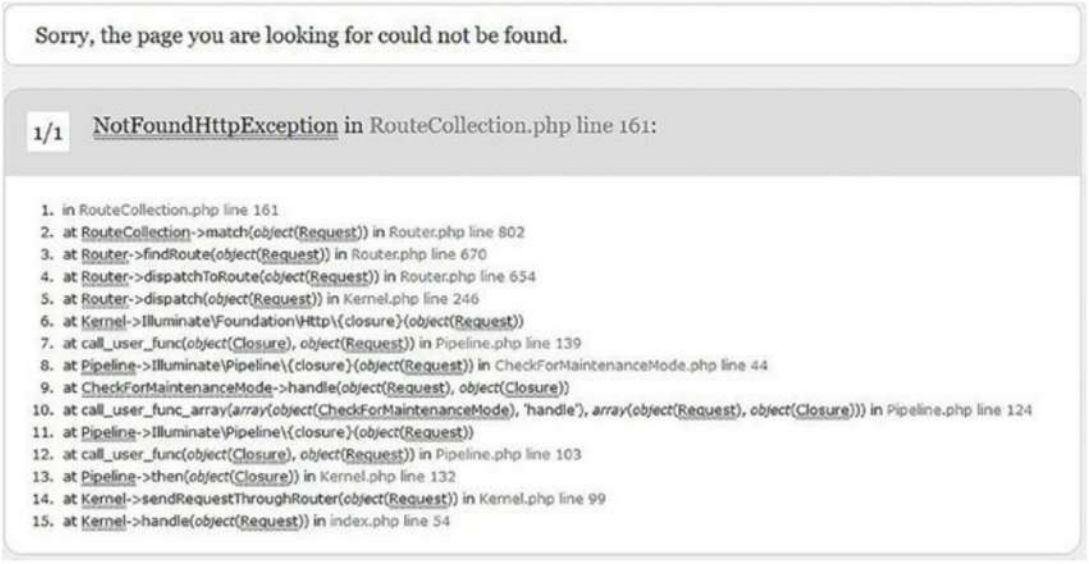
On peut rendre un paramètre optionnel en lui ajoutant un point d'interrogation, mais il ne doit pas être suivi par un paramètre obligatoire.

Copyright © 2016 Eyrolles.

Erreur d'exécution et contrainte de route

Dans mon double exemple précédent, lorsque je dis que le résultat est exactement le même, je mens un peu. Que se passe-t-il dans les deux cas pour l'URL *http://monsite.fr/4* ?

Dans le cas des trois routes, vous tombez sur une erreur.



Erreur d'exécution : la route n'existe pas.

Dans la version paramétrée en revanche, vous obtenez une réponse valide :

```
Je suis la page 4 !
```

C'est logique, parce qu'une route est trouvée. Toutefois, le paramètre accepte n'importe quelle valeur et pas seulement des nombres. Par exemple, avec l'URL *http://monsite.fr/nimportequoi*, vous obtenez :

```
Je suis la page nimportequoi !
```

Vous avouerez que ce n'est pas très heureux !

Pour éviter ce genre de désagrément, il faut contraindre le paramètre à n'accepter que certaines valeurs. On réalise cela à l'aide d'une expression régulière :

```
<?php
Route::get('{n}', function($n) {
  return 'Je suis la page ' . $n . ' !';
})->where('n', '[1-3]');
```

Copyright © 2016 Eyrolles.

Maintenant, je peux affirmer que les comportements sont identiques ! Il nous restera à régler le problème des routes non prévues. Nous verrons cela dans un prochain chapitre.

Une route nommée

Il est parfois utile de nommer une route, par exemple pour générer une URL ou pour effectuer une redirection. La syntaxe pour nommer une route est la suivante :

```
<?php
Route::get('/', ['as'=>'home', function()
{
  return 'Je suis la page d\'accueil !';
}]);
```

Nous présenterons des cas d'utilisation de routes nommées dans les prochains chapitres.

Les façades

Laravel propose de nombreuses façades pour simplifier la syntaxe. Vous les trouvez toutes déclarées dans le fichier `config/app.php` :

```
<?php
'aliases'=>[
  'App'     =>Illuminate\Support\Facades\App::class,
  'Artisan'=>Illuminate\Support\Facades\Artisan::class,
  'Auth'    =>Illuminate\Support\Facades\Auth::class,
  'Blade'   =>Illuminate\Support\Facades\Blade::class,
  'Cache'   =>Illuminate\Support\Facades\Cache::class,
  'Config'  =>Illuminate\Support\Facades\Config::class,
  'Cookie'  =>Illuminate\Support\Facades\Cookie::class,
  'Crypt'   =>Illuminate\Support\Facades\Crypt::class,
  'DB'      =>Illuminate\Support\Facades\DB::class,
  'Eloquent'=>Illuminate\Database\Eloquent\Model::class,
  'Event'   =>Illuminate\Support\Facades\Event::class,
  'File'    =>Illuminate\Support\Facades\File::class,
  'Gate'    =>Illuminate\Support\Facades\Gate::class,
  'Hash'    =>Illuminate\Support\Facades\Hash::class,
  'Lang'    =>Illuminate\Support\Facades\Lang::class,
  'Log'     =>Illuminate\Support\Facades\Log::class,
  'Mail'    =>Illuminate\Support\Facades\Mail::class,
  'Password'=>Illuminate\Support\Facades\Password::class,
  'Queue'    =>Illuminate\Support\Facades\Queue::class,
  'Redirect'=>Illuminate\Support\Facades\Redirect::class,
```

Copyright © 2016 Eyrolles.

```
  'Redis'    =>Illuminate\Support\Facades\Redis::class,
  'Request'  =>Illuminate\Support\Facades\Request::class,
  'Response'=>Illuminate\Support\Facades\Response::class,
  'Route'    =>Illuminate\Support\Facades\Route::class,
  'Schema'   =>Illuminate\Support\Facades\Schema::class,
  'Session'  =>Illuminate\Support\Facades\Session::class,
  'Storage'  =>Illuminate\Support\Facades\Storage::class,
  'URL'      =>Illuminate\Support\Facades\URL::class,
  'Validator'=>Illuminate\Support\Facades\Validator::class,
  'View'     =>Illuminate\Support\Facades\View::class,
  'Form'     =>Collective\Html\FormFacade::class,
  'Html'     =>Collective\Html\HtmlFacade::class,
],
```

Vous trouvez dans ce tableau le nom de la façade et la classe qui la met en place. Par exemple, pour les routes, on a la façade `Route` qui correspond à la classe `Illuminate\Support\Facades\Route`. Examinons cette dernière :

```
<?php
namespace Illuminate\Support\Facades;
/**
 * @see \Illuminate\Routing\Router
 */
class Route extends Facade
{
  /**
   * Get the registered name of the component.
   *
   * @return string
   */
  protected static function getFacadeAccessor()
  {
    return 'router';
  }
}
```

On se contente de retourner `'router'`. Il faut aller voir dans le fichier `Illuminate\Routing\RoutingServiceProvider` pour trouver l'enregistrement correspondant :

```
<?php
/**
 * Register the router instance.
 *
 * @return void
 */
protected function registerRouter()
{
  $this->app['router']=$this->app->share(function ($app) {
    return new Router($app['events'], $app);
  });
}
```

Copyright © 2016 Eyrolles.

Les fournisseurs (*providers*) permettent d'enregistrer des composants dans le conteneur de Laravel. Ici, on déclare `'router'` et on voit qu'on crée une instance de la classe `Router` (`new Router...`). Le nom complet est `Illuminate\Routing\Router`. Si vous allez voir cette classe, vous trouvez les méthodes qu'on a utilisées dans ce chapitre, par exemple `get` :

```
<?php
/**
 * Register a new GET route with the router.
 *
 * @param  string $uri
 * @param  \Closure|array|string $action
 * @return \Illuminate\Routing\Route
 */
public function get($uri, $action)
{
  return $this->addRoute(['GET', 'HEAD'], $uri, $action);
}
```

Autrement dit, si j'écris en utilisant la façade :

```
<?php
Route::get('/', function() {return 'Coucou';});
```

j'obtiens le même résultat que si j'écris en allant chercher le routeur dans le conteneur :

```
<?php
$this->app['router']->get('/', function() {return 'Coucou';});
```

ou en utilisant un *helper* :

```
<?php
app('router')->get('/', function() {return 'Coucou';});
```

La différence est que la première syntaxe est plus simple et intuitive, mais certains n'aiment pas trop ce genre d'appel statique.

En résumé

- Laravel possède un fichier `.htaccess` pour simplifier l'écriture des URL.
- Le système de routage est simple et explicite.
- On peut prévoir des paramètres dans les routes.
- On peut contraindre un paramètre à correspondre à une expression régulière.
- On peut nommer une route pour faciliter la génération des URL et les redirections.

Copyright © 2016 Eyrolles.

- Il faut prévoir de gérer toutes les URL, même celles qui n'ont aucune route prévue.
- Laravel est équipé de nombreuses façades qui simplifient la syntaxe.
- Il existe aussi des fonctions *helpers* pour simplifier la syntaxe.

Copyright © 2016 Eyrolles.

4 Les réponses

Nous avons expliqué précédemment comment la requête qui arrive est traitée par les routes. Voyons maintenant les réponses que nous pouvons renvoyer au client. Nous allons présenter le système des vues de Laravel avec la possibilité de transmettre des paramètres. Nous décrirons aussi comment créer des templates avec l'outil `Blade`.

Construire une réponse

Codes des réponses

Dans le protocole HTTP, il existe, classés par grandes catégories, des codes pour spécifier les réponses. Voici les trois principaux :

- 200 : requête exécutée avec succès ;
- 404 : l'adresse demandée n'a pas été trouvée ;
- 500 : erreur sur le serveur.

Dans les exemples utilisés dans le chapitre précédent sur les routes, je n'ai pas précisé de code ; je me suis contenté de retourner un texte au client. Ce dernier veut quelque chose d'explicite, alors que les moteurs de recherche savent interpréter les codes.

En plus du simple `return` pour renvoyer la réponse, je peux utiliser l'*helper* `response` et préciser le code :

```
<?php
Route::get('{n}', function($n) {
  return response('Je suis la page ' . $n . ' !', 200);
})->where('n', '[1-3]');
```

Maintenant, je retourne une véritable réponse HTTP avec le code correspondant.

Copyright © 2016 Eyrolles.

Il existe la façade `Response` pour les réponses. On peut donc écrire de façon équivalente :

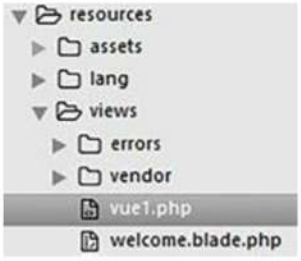
```
<?php
return Response::make('Je suis la page ' . $n . ' !', 200);
```

Vues

Dans une application réelle, vous retournerez rarement la réponse directement à partir d'une route ; vous passerez au moins par une vue. Dans sa version la plus simple, une vue est un simple fichier avec du code HTML :

```
<!doctype html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>Ma première vue</title>
<body>
  Je suis une vue !
</body>
</html>
```

Il faut enregistrer cette vue (j'ai choisi le nom `vue1`) dans le dossier `resources/views` avec l'extension `php`.



La vue dans le dossier des vues

Même si vous n'écrivez que du code HTML dans une vue, vous devez l'enregistrer avec l'extension `.php`.

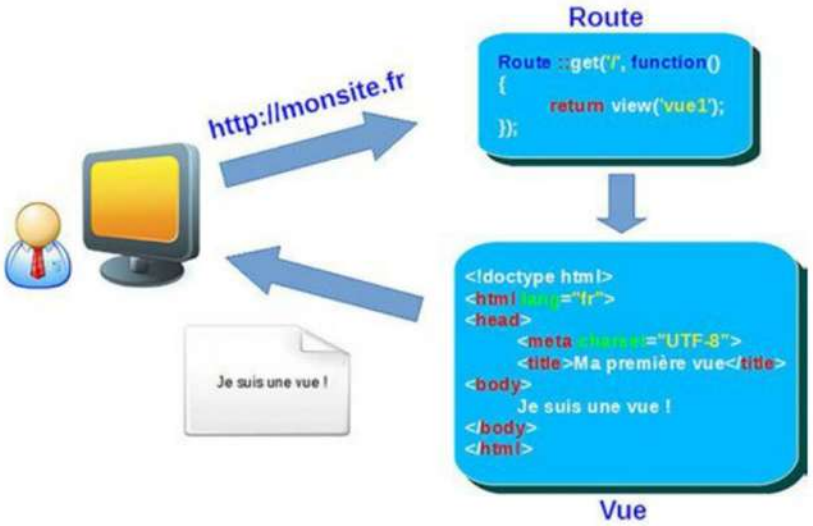
On peut appeler cette vue à partir d'une route avec le code suivant :

```
<?php
Route::get('/', function()
{
  return view('vue1');
});
```

Copyright © 2016 Eyrolles.

Je vous rappelle la belle sémantique de Laravel qui se lit comme de la prose : je retourne (`return`) une vue (`view`) à partir du fichier de vue `vue1`.

Voici une illustration du processus.



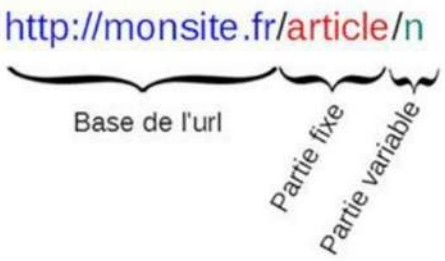
Le cycle de requête avec une vue

La vue paramétrée

URL

En général, on doit transmettre des informations à une vue. Voyons à présent comment mettre cela en place. Supposons que nous voulions répondre à une requête de la forme suivante, où le paramètre `n` doit prendre une valeur numérique :

```
http://monsite.fr/article/n
```



Constitution de l'URL

Voyons comment cette URL est constituée :

- la base de l'URL est constante pour le site, quelle que soit la requête ;

Copyright © 2016 Eyrolles.

- la partie fixe ici correspond aux articles ;
- la partie variable correspond au numéro de l'article désiré.

Route

Il nous faut une route pour intercepter ces URL :

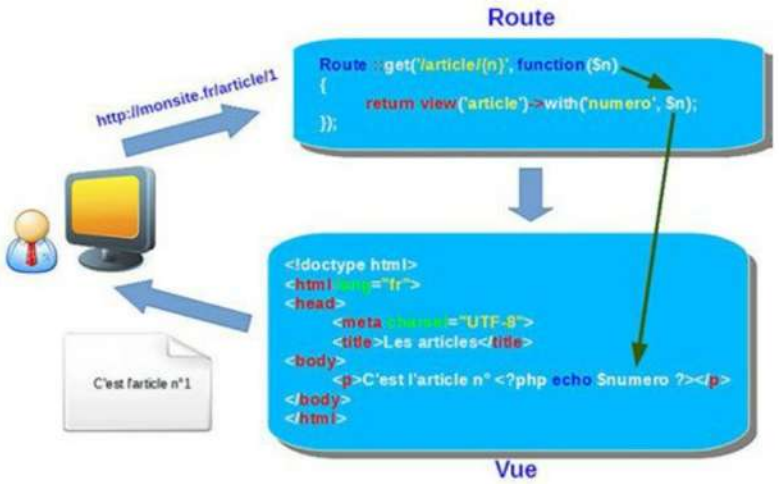
```
<?php
Route::get('article/{n}', function($n) {
  return view('article')->with('numero', $n);
})->where('n', '[0-9]+');
```

Vue

Il ne nous reste plus qu'à créer la vue `article.php` dans le dossier `resources/views` :

```
<!doctype html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>Les articles</title>
<body>
  <p>C'est l'article n° <?php echo $numero ?></p>
</body>
</html>
```

Pour récupérer le numéro de l'article, on utilise la variable `$numero`. Voici une schématisation du fonctionnement.



Demande de l'article numéro 1

Il existe une méthode « magique » pour transmettre un paramètre, par exemple la variable `numero` de notre exemple. Il suffit de concaténer le nom de la variable au mot clé `with` :

Copyright © 2016 Eyrolles.

```
return view('article')->withNumero($n);
```

On peut aussi transmettre un tableau comme deuxième paramètre :

```
<?php
return view('article', ['numero'=>$n]);
```

Simplifier la syntaxe

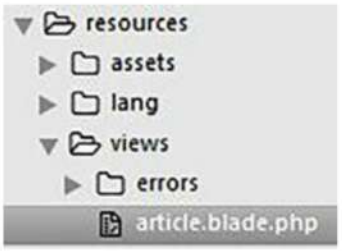
Laravel possède un moteur de templates élégant nommé `Blade`, qui facilite beaucoup de choses. La première est de nous simplifier la syntaxe. Prenons par exemple la ligne suivante :

```
<p>C'est l'article n° <?php echo $numero ?></p>
```

On peut l'écrire avec la syntaxe de `Blade` :

```
<p>C'est l'article n° {{$numero}}</p>
```

Tout ce qui se trouve entre les doubles accolades est interprété comme du code PHP. Cependant, pour que cela fonctionne, il faut indiquer à Laravel qu'on veut utiliser `Blade` pour cette vue.



Une vue activée pour Blade

Il suffit d'ajouter `blade` au nom du fichier, avant l'extension `php`. Vous pouvez tester l'exemple précédent avec ces modifications et vous verrez que tout fonctionne parfaitement avec une syntaxe épurée.

Il y a aussi la version avec la syntaxe `{!!...!!}`. La différence est que le texte entre les doubles accolades est échappé ou purifié. C'est une mesure de sécurité parce qu'un utilisateur pourrait très bien mettre du code malveillant dans l'URL.

Copyright © 2016 Eyrolles.

Template

Une fonction fondamentale de `Blade` est de permettre le *templating*, c'est-à-dire de factoriser le code de présentation. Poursuivons notre exemple en complétant notre application avec une autre route chargée d'intercepter des URL pour des factures, ainsi que sa vue :

Route

```
<?php
Route::get('facture/{n}', function($n) {
  return view('facture')->withNumero($n);
})->where('n', '[0-9]+');
```

Vue

```
<!doctype html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>Les factures</title>
<body>
  <p>C'est la facture n° {{$numero}}</p>
</body>
</html>
```

On se rend compte que cette vue est pratiquement la même que celle pour les articles. Il serait intéressant de placer le code commun dans un fichier. C'est justement le but d'un *template*.

Template

```
<!doctype html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>@yield('titre')</title>
<body>
  @yield('contenu')
</body>
</html>
```

J'ai repris le code commun et prévu deux emplacements repérés par le mot-clé `@yield` et nommés `titre` et `contenu`. Il suffit maintenant de modifier les deux vues.

Copyright © 2016 Eyrolles.

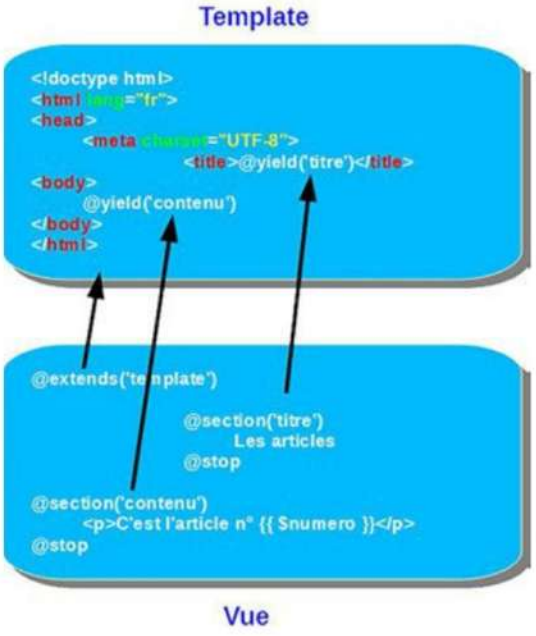
Vue pour les articles

```
@extends('template')

@section('titre')
  Les articles
@endsection

@section('contenu')
  <p>C'est l'article n° {{$numero}}</p>
@endsection
```

Vue pour les factures

```
@extends('template')

@section('titre')
  Les factures
@endsection

@section('contenu')
  <p>C'est la facture n° {{$numero}}</p>
@endsection
```

Dans un premier temps, on dit qu'on veut utiliser le template avec `@extends` et le nom du template. Ensuite, on remplit les zones prévues dans le template grâce à la syntaxe `@section` en précisant le nom de l'emplacement et en fermant avec `@endsection` (l'ancienne syntaxe `@stop` est encore fonctionnelle). Voici un schéma pour bien visualiser tout cela avec les articles.



Fonctionnement du template

Copyright © 2016 Eyrolles.

Au niveau du dossier des vues, on a donc les trois fichiers.

Dossier des vues

Lorsqu'elles deviendront nombreuses, on organisera nos vues dans des dossiers. Vous pouvez d'ailleurs remarquer qu'il en existe déjà plusieurs dans l'installation de base.

Les redirections

Souvent, il ne faut pas envoyer directement la réponse, mais rediriger sur une autre URL. Pour réaliser cela, on a l'*helper* `redirect` :

```
<?php
return redirect('facture');
```

Ici, on redirige sur l'URL `facture`. On peut aussi rediriger sur une route nommée :

```
<?php
return redirect()->route('facture');
```

Ici on redirige sur la route nommée `facture`.

On verra de nombreux exemples de redirections dans les prochains chapitres.

En résumé

- Laravel offre la possibilité de créer des vues.
- Il est facile de transmettre des paramètres aux vues.
- L'outil `Blade` permet de créer des templates et d'optimiser ainsi le code des vues.
- On peut facilement effectuer des redirections.

Copyright © 2016 Eyrolles.

5 Les contrôleurs

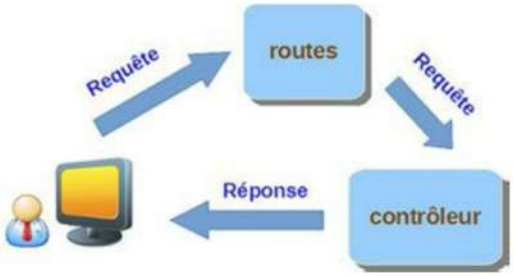
Nous avons expliqué le cycle d'une requête depuis son arrivée, son traitement par les routes et sa réponse avec des vues qui peuvent être améliorées par `Blade`. Avec tous ces éléments, vous pourriez réaliser un site web complet, mais Laravel offre encore de nombreux outils performants.

Pour bien organiser son code dans une application Laravel, il faut savoir répartir les tâches. Dans les exemples précédents, j'ai renvoyé une vue à partir d'une route ; vous ne ferez jamais cela dans une application réelle (même si personne ne vous empêche de le faire !). Les routes sont juste un système d'aiguillage pour trier les requêtes qui arrivent. Mais alors qui s'occupe de la suite ? Eh bien, ce sont les contrôleurs, le sujet de ce chapitre.

L'utilité des contrôleurs

Rôle

La tâche d'un contrôleur est de réceptionner une requête (qui a déjà été triée par une route) et de définir la réponse appropriée, rien de moins et rien de plus.



Traitement de la requête par un contrôleur

Copyright © 2016 Eyrolles.

Constitution

Pour créer un contrôleur, nous allons utiliser `Artisan`, la boîte à outils de Laravel. Dans la console, entrez la commande suivante :

```
php artisan make:controller WelcomeController
```

Le contrôleur créé

Voici le code du contrôleur créé :

```
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class WelcomeController extends Controller
{
  //
}
```

Vous allez ajouter la méthode `index` :

```
<?php
...
class WelcomeController extends Controller
{
  public function index()
  {
    return view('welcome');
  }
}
```

Analysons un peu ce code :

- on trouve en premier l'espace de nom ;
- le contrôleur hérite de la classe `Controller` qui se trouve dans le même dossier et qui permet de factoriser des actions communes à tous les contrôleurs ;
- on trouve enfin une méthode `index` qui renvoie la vue `welcome` dont nous avons déjà parlé.

Copyright © 2016 Eyrolles.

Liaison avec les routes

Comment s'effectue la liaison entre les routes et les contrôleurs ? Ouvrez le fichier des routes et entrez ce code :

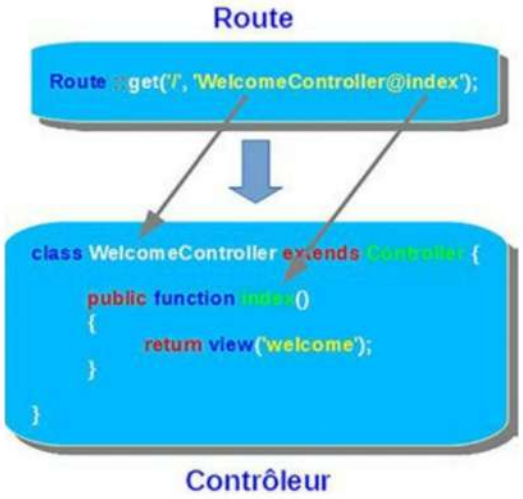
```
<?php
Route::get('/', 'WelcomeController@index');
```

Maintenant, avec l'URL de base, vous devez retrouver la page d'accueil de Laravel.

Laravel 5

Page d'accueil

Voici une visualisation de la liaison entre la route et le contrôleur.



Liaison entre la route et le contrôleur

On voit qu'au niveau de la route, il suffit de désigner le nom du contrôleur et celui de la méthode.

Si vous êtes attentif au code, vous avez sans doute remarqué qu'au niveau de la route on ne spécifie pas l'espace de noms du contrôleur ; on peut légitimement se demander comment on le retrouve. Laravel nous simplifie la syntaxe en ajoutant automatiquement l'espace de noms.

Si vous devez placer vos contrôleurs dans un autre espace de noms, il faut intervenir sur la variable `$namespace` dans le fichier `App\Providers\RouteServiceProvider` :

Copyright © 2016 Eyrolles.

```
<?php
protected $namespace='App\Http\Controllers';
```

Cette valeur constitue la base de référence des espaces de noms.

Route nommée

De la même manière que nous pouvons nommer une route classique, on peut aussi donner un nom à une route qui pointe vers une méthode de contrôleur :

```
<?php
Route::get('/', ['uses'=>'WelcomeController@index', 'as'=>'home']);
```

Ici, on nomme `home` la route vers la méthode `index` du contrôleur `WelcomeController` pour l'URL de base.

L'utilisation d'un contrôleur

Voyons un exemple pratique de mise en œuvre d'un contrôleur. On conservera notre exemple avec les articles, mais maintenant traité avec un contrôleur. On conserve le même template et les mêmes vues.

Le template et les vues

On crée un contrôleur (entraînez-vous à utiliser `Artisan`) pour les articles :

```
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class ArticleController extends Controller
{
   public function show($n)
  {
    return view('article')->with('numero', $n);
  }
}
```

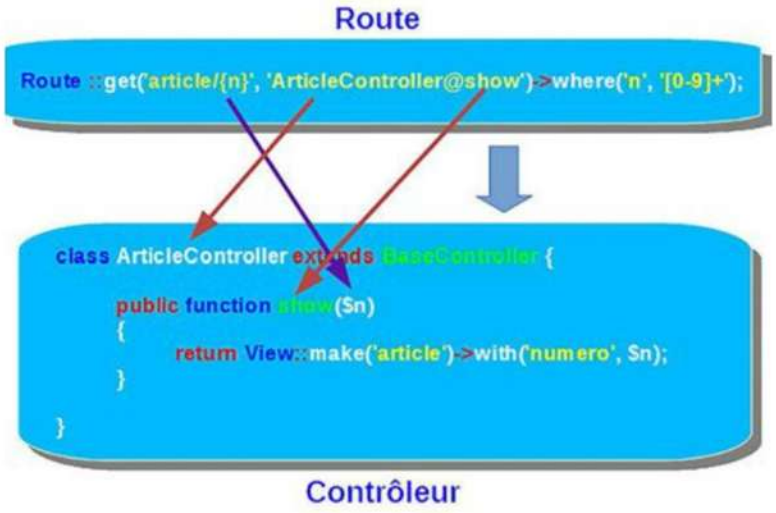
Copyright © 2016 Eyrolles.

Le dossier des contrôleurs

Dans ce contrôleur, la méthode `show` est chargée de générer la vue. Il ne nous reste plus qu'à créer la route :

```
<?php
Route::get('article/{n}', 'ArticleController@show')->where('n', '[0-9]+');
```

Voici une illustration du fonctionnement avec un contrôleur.



Les articles avec un contrôleur

Notez qu'on pourrait utiliser la méthode « magique » pour la transmission du paramètre à la vue :

```
<?php
return view('article')->withNumero($n);
```

En résumé

- Les contrôleurs servent à réceptionner les requêtes triées par les routes et à fournir une réponse au client.

Copyright © 2016 Eyrolles.

- `Artisan` facilite la création d'un contrôleur.
- Il est facile d'appeler une méthode de contrôleur à partir d'une route.
- On peut nommer une route qui pointe vers une méthode de contrôleur.

Copyright © 2016 Eyrolles.

6 Les entrées

Dans bien des circonstances, le client envoie des informations au serveur. La situation la plus générale est celle d'un formulaire. Nous allons voir dans ce chapitre comment créer facilement un formulaire avec Laravel et réceptionner les entrées. Ce faisant, nous améliorerons notre compréhension du routage.

Scénario et routes

Nous allons envisager un petit scénario avec une demande de formulaire de la part du client, sa soumission et son traitement.

Scénario

On aura donc besoin de deux routes :

- une pour la demande du formulaire avec une méthode `get` ;
- une pour la soumission du formulaire avec une méthode `post`.

On les crée dans le fichier `app/route.php` :

Copyright © 2016 Eyrolles.

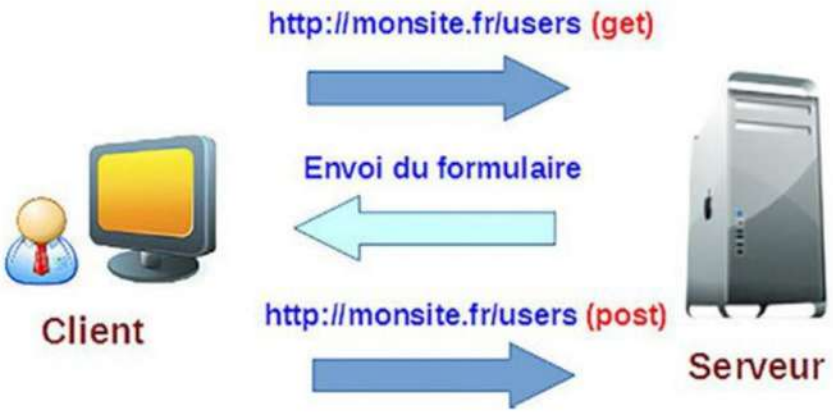
```php
<?php
Route::get('users', 'UsersController@getInfos');
Route::post('users', 'UsersController@postInfos');
```

Jusque-là, on avait vu seulement des routes avec le verbe `get` ; on a maintenant aussi une route avec le verbe `post`.

Laravel autorise d'autres verbes comme `put` et `delete`, ou plusieurs verbes pour une même route avec `match`, ou même tous les verbes avec `any`.

On a la même URL *http://monsite.fr/users*, mais selon le cas le verbe est soit `get`, soit `post`.

Voici le scénario schématisé avec les URL.



Les URL du scénario

Le middleware

Je parlerai plus en détail des *middlewares* dans un prochain chapitre. Pour le moment, on se contentera de savoir que c'est du code qui est activé à l'arrivée de la requête (ou à son départ) pour effectuer un traitement. C'est pratique pour arrêter par exemple directement la requête s'il y a un problème de sécurité.

Laravel peut servir comme application web ou comme api. Dans le premier cas, on a besoin de gérer :

- les cookies ;
- une session ;
- la protection CSRF (dont je parle plus loin dans ce chapitre).

Regardez dans le fichier `app/Http/Kernel.php` :

Copyright © 2016 Eyrolles.

```
<?php
protected $middlewareGroups=[
  'web'=>[
    \App\Http\Middleware\EncryptCookies::class,
    \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    \App\Http\Middleware\VerifyCsrfToken::class,
  ],

  'api'=>[
    'throttle:60,1',
  ],
];
```

On trouve les deux *middlewares* de groupes web et api. On voit que, dans le premier cas, on active bien les cookies, les sessions et la vérification CSRF.

Par défaut, toutes les routes que vous entrez dans le fichier `app/Http/routes.php` sont incluses dans le groupe `web`. Si vous regardez dans le provider `app/Providers/RouteServiceProvider.php`, vous trouvez cette inclusion :

```
<?php
protected function mapWebRoutes(Router $router)
{
    $router->group([
        'namespace' => $this->namespace, 'middleware' => 'web',
    ], function ($router) {
        require app_path('Http/routes.php');
    });
```

Le formulaire

Pour faire les choses correctement, nous allons prévoir un template `resources/views/template.blade.php` :

```
<!doctype html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
<body>
  @yield('contenu')
</body>
</html>
```

Il faut également une vue `resources/views/infos.blade.php` qui utilise ce template :

Copyright © 2016 Eyrolles.

```
@extends('template')

@section('contenu')
  {!! Form::open(['url'=>'users']) !!}
    {!! Form::label('nom', 'Entrez votre nom : ') !!}
    {!! Form::text('nom') !!}
    {!! Form::submit('Envoyer !') !!}
  {!! Form::close() !!}
@endsection
```

Ce code mérite quelques explications. Pour créer un formulaire, il faut commencer par l'ouvrir :

```
Form::open(['url'=>'users'])
```

La sémantique est simple : on veut ouvrir (`open`) un formulaire (`Form`) et le faire pointer vers l'URL `users`.

Le formulaire contient :

- une étiquette (`label`) :

```
Form::label('nom', 'Entrez votre nom : ')
```

- un contrôle de type `text` qui se nomme `nom` :

```
Form::text('nom')
```

- un bouton de soumission (`submit`) avec le texte ***Envoyer !*** :

```
Form::submit('Envoyer !')
```

Finalement, on ferme (`close`) le formulaire :

```
Form::close()
```

Le code généré est alors le suivant :

```
<form method="POST" action="http://monsite.fr/users" accept-charset="UTF-8">
  <input name="_token" type="hidden"
value="pV1vWWdUqFDfYsBjKag43C3NvzbIC0lHtMnv9BpI">
  <label for="nom">Entrez votre nom : </label>
  <input name="nom" type="text" id="nom">
  <input type="submit" value="Envoyer !">
</form>
```

Copyright © 2016 Eyrolles.

Voici quelques remarques sur ce code :

- la méthode par défaut est `post` ; on n'a pas eu besoin de le préciser ;
- l'action est bien construite ;
- il y a un contrôle caché (`_token`) destiné à la protection CSRF dont je parlerai plus loin ;
- l'étiquette est bien créée avec son attribut `for` ;
- le contrôle de texte est du bon type avec le bon nom. Il a en plus un `id` pour qu'il fonctionne avec son étiquette ;
- le bouton de soumission a été généré avec son texte.

Le résultat sera un formulaire sans fioriture.

Le formulaire créé

Si votre formulaire ne se génère pas, c'est que vous n'avez peut-être pas chargé le composant `Laravelcollective\Html` comme nous l'avons vu dans le chapitre sur l'installation.

Vous n'êtes pas obligé d'utiliser le composant `Laravelcollective\Html` pour créer des formulaires, mais je vous y encourage parce qu'il simplifie le codage et que je l'utilise tout au long de cet ouvrage. Par exemple, pour créer le formulaire sans l'utiliser, il nous faudrait écrire ceci :

```
@extends('template')

@section('contenu')
  <form method="POST" action="{!! url('users') !!}" accept-charset="UTF-8">
    {!! csrf_field() !!}
    <label for="nom">Entrez votre nom : </label>
    <input name="nom" type="text" id="nom">
    <input type="submit" value="Envoyer !">
  </form>
@endsection
```

Le contrôleur

Il ne nous manque plus que le contrôleur pour faire fonctionner tout cela :

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Http\Requests;
```

Copyright © 2016 Eyrolles.

```php
use App\Http\Controllers\Controller;

class UsersController extends Controller
{
  public function getInfos()
  {
    return view('infos');
  }

  public function postInfos(Request $request)
  {
    return 'Le nom est ' . $request->input('nom');
  }
}
```

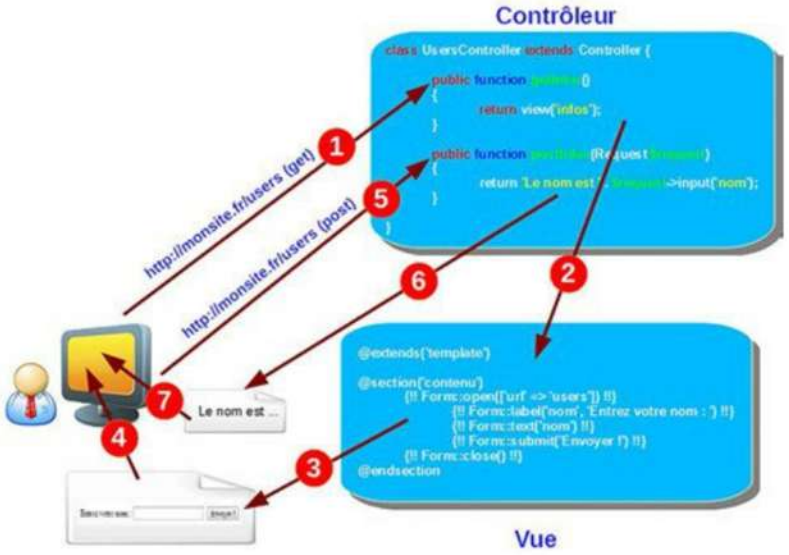
Mon contrôleur possède deux méthodes :

- `getInfos` qui reçoit l'URL *http://monsite.fr/users* avec le verbe `get` et qui retourne le formulaire ;
- `postInfos` qui reçoit l'URL *http://monsite.fr/users* avec le verbe `post` et qui traite les entrées.

Pour la première méthode, il n'y a rien de nouveau et je vous renvoie aux chapitres précédents si quelque chose ne vous paraît pas clair.

Dans la seconde méthode, on veut récupérer l'entrée du client. Encore une fois, la syntaxe est limpide : on veut dans la requête (`$request`) récupérer celle des entrées (`input`) qui s'appelle `nom`.

Si vous faites fonctionner tout cela, vous devez au final obtenir l'affichage du nom saisi. Voici une schématisation du fonctionnement qui exclut les routes pour simplifier.



Le scénario en action

Copyright © 2016 Eyrolles.

1. Le client envoie la requête de demande du formulaire, qui est transmise au contrôleur par la route (non représentée sur le schéma).
2. Le contrôleur crée la vue `infos`.
3. La vue `infos` crée le formulaire.
4. Le formulaire est envoyé au client.
5. Le client soumet le formulaire. Le contrôleur reçoit la requête de soumission par l'intermédiaire de la route (non représentée sur le schéma).
6. Le contrôleur construit la réponse.
7. La réponse est envoyée au client.

Il existe la façade `Request` pour récupérer les entrées :

```
<?php
Request::input('nom')
```

La protection CSRF

On a vu que le formulaire généré par Laravel comporte un contrôle caché avec une valeur particulière :

```
<input name="_token" type="hidden"
value="pV1vWWdUqFDfYsBjKag43C3NvzbIC0lHtMnv9BpI">
```

À quoi cela sert-il ? Tout d'abord, CSRF signifie *Cross-Site Request Forgery*. C'est une attaque qui consiste à faire envoyer par un client une requête à son insu. Cette attaque est relativement simple à mettre en place et consiste à envoyer à un client authentifié sur un site un script dissimulé (dans une page web ou un courriel) pour lui faire accomplir une action à son insu.

Pour se prémunir contre ce genre d'attaque, Laravel génère un *token* aléatoire associé au formulaire, qui est vérifié à la soumission du formulaire pour être sûr de l'origine.

Vous vous demandez peut-être où se trouve ce middleware `CSRF`. Il est bien rangé dans le dossier `App/Http/Middleware` :

Copyright © 2016 Eyrolles.

Le middleware CSRF

Pour tester l'efficacité de cette vérification, essayez un envoi de formulaire sans le *token* en modifiant la vue comme suit (adaptez la valeur de l'action selon votre contexte) :

```
@extends('template')

@section('contenu')
  <form method="POST" action="http://monsite.fr/users" accept-
charset="UTF-8">
    <label for="nom">Entrez votre nom : </label>
    <input name="nom" type="text" id="nom">
    <input type="submit" value="Envoyer !">
  </form>
@endsection
```

Vous tomberez sur l'erreur suivante à la soumission.

Erreur dans la vérification du token

En résumé

- Laravel permet de créer des routes avec différents verbes : `get`, `post`...
- Un formulaire est facile à créer avec la classe `Form`.
- Les entrées du client sont récupérées dans la requête.
- On peut se prémunir contre les attaques CSRF. Cette défense est mise en place automatiquement par Laravel.

Copyright © 2016 Eyrolles.

7 La validation

Nous avons vu dans le chapitre précédent un scénario mettant en oeuvre un formulaire. Nous n'avons imposé aucune contrainte sur les valeurs transmises. Dans une application réelle, il est toujours nécessaire de vérifier que ces valeurs correspondent à ce qu'on attend. Par exemple, un nom doit comporter uniquement des caractères alphabétiques et avoir une longueur maximale, une adresse électronique doit correspondre à un certain format.

Il faut donc mettre en place des règles de validation. En général, on procède à une première validation côté client pour éviter de faire des allers-retours avec le serveur. Néanmoins, quelle que soit la pertinence de cette validation côté client, elle ne dispense pas d'une validation côté serveur.

On ne doit jamais faire confiance à des données qui arrivent sur le serveur.

Dans l'exemple de ce chapitre, je ne prévoirai pas de validation côté client, d'une part ce n'est pas mon propos, d'autre part elle masquerait la validation côté serveur pour les tests.

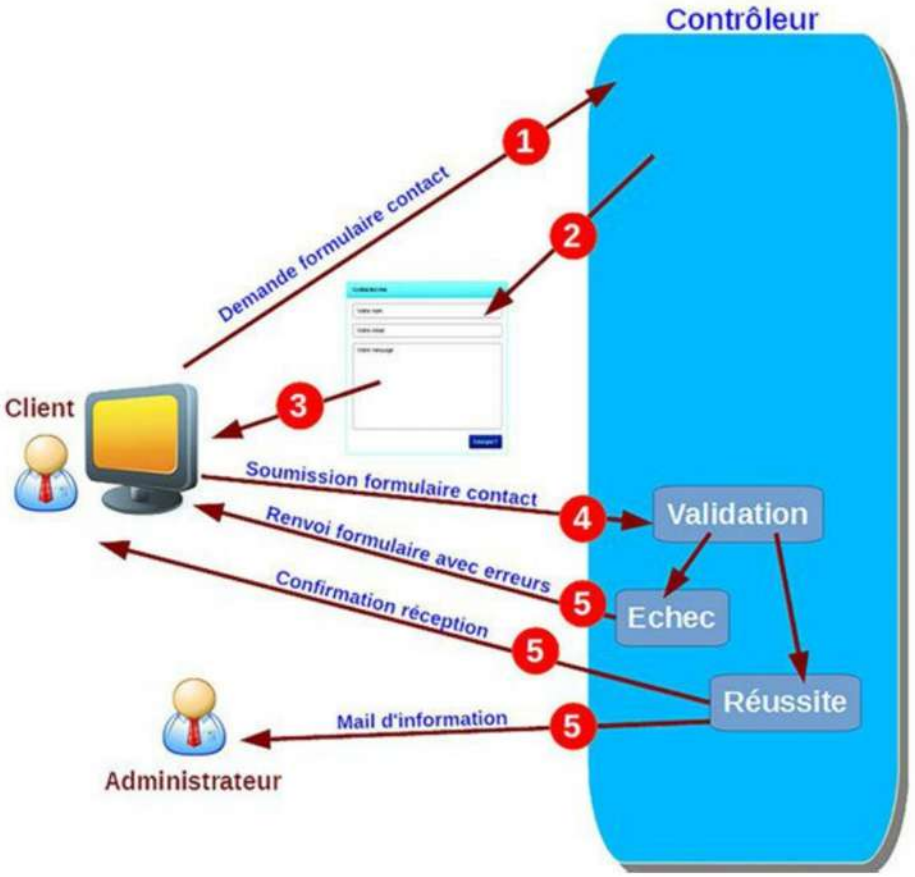
Scénario et routes

Voici le scénario que je vous propose pour ce chapitre :

- le client demande le formulaire de contact ;
- le contrôleur construit, puis envoie le formulaire ;
- le client remplit le formulaire et le soumet ;

Copyright © 2016 Eyrolles.

- le contrôleur teste la validité des informations :
  - en cas d'échec, le contrôleur renvoie le formulaire au client en l'informant des erreurs et en conservant ses entrées correctes ;
  - en cas de réussite, le contrôleur envoie un message de confirmation au client et un courriel à l'administrateur.



Scénario de prise de contact

On a donc besoin de deux routes :

```
<?php
Route::get('contact', 'ContactController@getForm');
Route::post('contact', 'ContactController@postForm');
```

On aura une seule URL (avec verbe `get` pour demander le formulaire et verbe `post` pour le soumettre) :

```
http://monsite.fr/contact
```

Copyright © 2016 Eyrolles.

Les vues

Template

Pour ce chapitre, je vais créer un template réaliste avec Bootstrap pour alléger le code (`resources/views/template.blade.php`) :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Mon joli site</title>
    {!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/
bootstrap.min.css') !!}
    {!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/
bootstrap-theme.min.css') !!}
    <!--[if lt IE 9]>
      {{ Html::style('https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.
js') }}
      {{ Html::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.
min.js') }}
    <![endif]-->
    <style> textarea {resize:none;} </style>
  </head>
  <body>
    @yield('contenu')
  </body>
</html>
```

Je rappelle que, dans `Blade`, il y a deux syntaxes : la double accolade permet de sécuriser le code en échappant les caractères spéciaux alors que `{!! !!}` n'effectue aucun traitement et doit donc être utilisé avec prudence.

Pour la génération des liens vers les bibliothèques CSS, j'ai utilisé la classe `Html` avec sa méthode `style`. Il y a un certain nombre de méthodes pratiques dans cette classe que nous découvrirons petit à petit.

J'ai prévu l'emplacement `@yield` nommé `contenu` pour recevoir les pages du site ; pour notre exemple, on aura seulement la page de contact et celle de la confirmation.

Vue de contact

La vue de contact contiendra essentiellement un formulaire (`resources/views/contact.blade.php`) :

Copyright © 2016 Eyrolles.

```
@extends('template')

@section('contenu')
  <br>
  <div class="col-sm-offset-3 col-sm-6">
    <div class="panel panel-info">
      <div class="panel-heading">Contactez-moi</div>
      <div class="panel-body">
        {!! Form::open(['url'=>'contact']) !!}
          <div class="form-group {!! $errors->has('nom') ? 'has-
error' : '' !!}">
            {!! Form::text('nom', null, ['class'=>'form-control',
'placeholder'=>'Votre nom']) !!}
            {!! $errors->first('nom', '<small class="help-block">:message</
small>') !!}
          </div>
          <div class="form-group {!! $errors->has('email') ? 'has-
error' : '' !!}">
            {!! Form::email('email', null, ['class'=>'form-control',
'placeholder'=>'Votre email']) !!}
            {!! $errors->first('email', '<small class="help-block">:message</
small>') !!}
          </div>
          <div class="form-group {!! $errors->has('texte') ? 'has-
error' : '' !!}">
            {!! Form::textarea ('texte', null, ['class'=>'form-control',
'placeholder'=>'Votre message']) !!}
            {!! $errors->first('texte', '<small class="help-block">:message</
small>') !!}
          </div>
          {!! Form::submit('Envoyer !', ['class'=>'btn btn-info pull-
right']) !!}
        {!! Form::close() !!}
      </div>
    </div>
  </div>
@endsection
```

Cette vue étend le template précédent et renseigne la section `"contenu"`. Je ne commente pas la mise en forme spécifique à Bootstrap. Le formulaire est généré avec la classe `Form` que nous avons déjà présentée.

La structure des méthodes pour générer les contrôles du formulaire est toujours la même. Prenons l'exemple de l'adresse électronique :

```
<?php
{!! Form::email('email', null, ['class'=>'form-control',
'placeholder'=>'Votre email']) !!}
```

On veut un élément de formulaire (`Form`) de type `email` avec le nom `'email'`, une valeur nulle et les attributs `'class'` et `'placeholder'` dont on précise la valeur.

Le bouton de soumission ne comporte évidemment pas de valeur et on a donc un paramètre de moins pour lui.

Copyright © 2016 Eyrolles.

En cas de réception du formulaire erroné, on reçoit une variable `$errors` qui contient un tableau avec comme clés les noms des contrôles et comme valeurs les textes identifiant les erreurs.

La variable `$errors` est générée systématiquement pour toutes les vues.

C'est pour cela que je teste la présence d'une erreur pour chaque contrôle en ajustant le style et en affichant le texte de l'erreur si nécessaire.

Avec la ligne suivante, j'obtiens une valeur booléenne qui m'indique s'il y a une erreur pour le nom :

```
$errors->has('nom')
```

Et avec la suivante, je récupère la description de la première erreur sur ce champ (il est possible qu'il y ait plusieurs erreurs, mais ce n'est pas la peine de les afficher toutes, surtout qu'en général on les hiérarchise et elles sont fréquemment dépendantes) :

```
$errors->first('nom')
```

Au départ, le formulaire se présente comme suit.

Le formulaire vierge

Après soumission et renvoi avec des erreurs, il peut changer d'aspect.

Copyright © 2016 Eyrolles.

Le formulaire avec des erreurs

Par défaut, les messages sont en anglais. Pour obtenir ces textes en français, vous devez récupérer les fichiers sur le site *https://github.com/caouecs/Laravel-lang*. Placez le dossier `fr` et son contenu dans le dossier `resources/lang`. Ensuite, changez la ligne suivante dans le fichier `config/app.php` : `'locale'=>'fr'`.

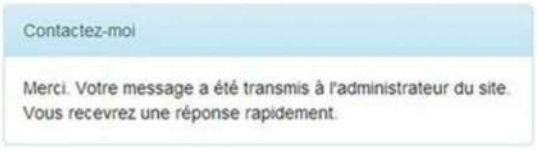
Les messages en français

Vue de confirmation

Pour la vue de confirmation (`resources/views/confirm.blade.php`), le code est plus simple et on utilise évidemment le même template :

Copyright © 2016 Eyrolles.

```
@extends('template')

@section('contenu')
  <br>
  <div class="col-sm-offset-3 col-sm-6">
    <div class="panel panel-info">
      <div class="panel-heading">Contactez-moi</div>
      <div class="panel-body">
        Merci. Votre message a été transmis à l'administrateur du site. Vous
recevrez une réponse rapidement.
      </div>
    </div>
  </div>
@endsection
```



La confirmation

Vue du courriel pour l'administrateur

Nous avons également besoin de créer une vue pour construire le courriel à destination de l'administrateur (`resources/views/email_contact.blade.php`) :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <h2>Prise de contact sur mon beau site</h2>
    <p>Réception d'une prise de contact avec les éléments suivants :</p>
    <ul>
      <li><strong>Nom</strong> : {{$nom}}</li>
      <li><strong>Email</strong> : {{$email}}</li>
      <li><strong>Message</strong> : {{$texte}}</li>
    </ul>
  </body>
</html>
```

On doit transmettre à cette vue les entrées de l'utilisateur.

La requête de formulaire

Nous allons de nouveau utiliser l'outil `Artisan`.

Copyright © 2016 Eyrolles.

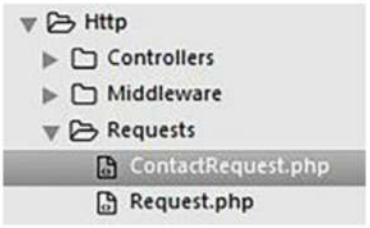
Pour les personnes allergiques à la console j'ai créé un package *https://github.com/bestmomo/nice-artisan* qui fournit les commandes d'`Artisan` avec une interface conviviale.

On y accède au niveau de la console :

```
php artisan
```

Parmi les nombreuses possibilités, nous allons utiliser `make:request` pour créer notre requête de formulaire :

```
php artisan make:request ContactRequest
Request created successfully.
```



La requête de formulaire

Voyons le code généré :

```
<?php
namespace App\Http\Requests;

use App\Http\Requests\Request;

class ContactRequest extends Request {
  /**
   * Determine if the user is authorized to make this request.
   *
   * @return bool
   */
  public function authorize()
  {
    return false;
  }

 /**
   * Get the validation rules that apply to the request.
   *
   * @return array
   */
 public function rules()
```

Copyright © 2016 Eyrolles.

```
  {
    return [
      //
    ];
  }
}
```

La classe comporte deux méthodes :

- `authorize` pour effectuer un contrôle de sécurité éventuel sur l'identité ou les droits de l'émetteur ;
- `rules` pour les règles de validation.

Adaptons le code pour notre cas :

```
<?php
namespace App\Http\Requests;

class ContactRequest extends Request {
  /**
   * Determine if the user is authorized to make this request.
   *
   * @return bool
   */
  public function authorize()
  {
    return true;
  }

  /**
   * Get the validation rules that apply to the request.
   *
   * @return array
   */
  public function rules()
  {
    return [
      'nom'=>'required|min:5|max:20|alpha',
      'email'=>'required|email',
      'texte'=>'required|max:250'
    ];
  }
}
```

Au niveau de la méthode `rules`, on retourne un tableau qui contient des clés correspondant aux champs du formulaire. Vous retrouvez le nom, l'adresse électronique et le texte. Les valeurs contiennent les règles de validation. Comme il y en a chaque fois plusieurs, elles sont séparées par le signe `|`. Voyons les différentes règles prévues :

- `required` : une valeur est requise, donc le champ ne doit pas être vide ;
- `min` : un nombre minimal de caractères est requis (par exemple, `min:5` signifie « au minimum 5 caractères ») ;
- `max` : un nombre maximal de caractères est requis ;

Copyright © 2016 Eyrolles.

- `alpha` : on n'accepte que les caractères alphabétiques ;
- `email` : la valeur doit être une adresse électronique valide.

Au niveau de la méthode `authorize`, je me suis contenté de renvoyer `true` parce que nous ne ferons pas de contrôle supplémentaire.

Il existe une règle `between:min,max` qui résume l'utilisation des deux règles `max:value` et `min:value`. Vous pouvez trouver toutes les règles disponibles dans la documentation : *https://laravel.com/docs/5.2/validation#available-validation-rules*. La liste est longue !

Le contrôleur

Voici le code du contrôleur `ContactController` :

```
<?php
namespace App\Http\Controllers;

use Mail;
use App\Http\Requests\ContactRequest;

class ContactController extends Controller {
  public function getForm()
  {
    return view('contact');
  }

  public function postForm(ContactRequest $request)
  {
    Mail::send('email_contact', $request->all(), function($message)
    {
      $message->to('monadresse@free.fr')->subject('Contact');
    });
    return view('confirm');
  }
}
```

La méthode `getForm` ne présente aucune nouveauté par rapport à ce qu'on a vu au chapitre précédent. On se contente de renvoyer la vue `contact` qui comporte le formulaire.

La méthode `postForm` nécessite quelques commentaires. Vous remarquez les paramètres de type `ContactRequest`. On injecte dans la méthode une instance de la classe `ContactRequest` qu'on a précédemment créée. Laravel permet ce genre d'injection de dépendances au niveau d'une méthode. Je reviendrai en détail sur cette possibilité dans un prochain chapitre.

Si la validation échoue parce qu'une règle n'est pas respectée, c'est la classe `ContactRequest` qui s'occupe de tout ; elle renvoie le formulaire en complétant les contrôles qui étaient corrects et crée une variable `$errors` pour transmettre les messages d'erreur qu'on utilise dans la vue. Vous n'avez rien d'autre à faire !

Copyright © 2016 Eyrolles.

J'ai utilisé la façade `Mail`, j'aurais pu aussi injecter dans la méthode un objet comme je l'ai fait pour la requête de formulaire. Nous verrons dans un chapitre ultérieur l'injection de dépendances avec d'autres cas d'utilisation.

Envoyer un courriel

En cas de réussite de la validation, on envoie un courriel à l'administrateur :

```
<?php
Mail::send('email_contact', $request->all(), function($message)
{
  $message->to('monadresse@free.fr')->subject('Contact');
});
```

Laravel utilise `SwiftMailer` pour accomplir cette tâche. La syntaxe est simple : on utilise la classe `Mail` pour envoyer (`send`) un e-mail avec le contenu de la vue `email_contact` à laquelle on transmet les éléments de la requête (`$request`) en précisant qu'on les veut tous (`all`). On envoie cet e-mail à (`to`) *monadresse@free.fr* avec comme sujet (`subject`) `'Contact'`.

Pour que l'envoi des courriels fonctionne, il faut renseigner les éléments suivants dans le fichier `.env` :

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.free.fr
MAIL_PORT=25
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

Il faut également ajouter la ligne qui suit dans `config/mail.php` :

```
'from'=>['address'=>'moi@free.fr', 'name'=>'Administrateur'],
```

Ici, la configuration correspond à mon hébergeur (Free) pour mes tests en local. Si vous avez un hébergeur différent, vous devrez évidemment adapter ces valeurs. Il en va de même si vous mettez un site en production sur un serveur.

Si tout se passe bien, vous devez recevoir un courriel semblable à celui de la figure suivante.

Copyright © 2016 Eyrolles.

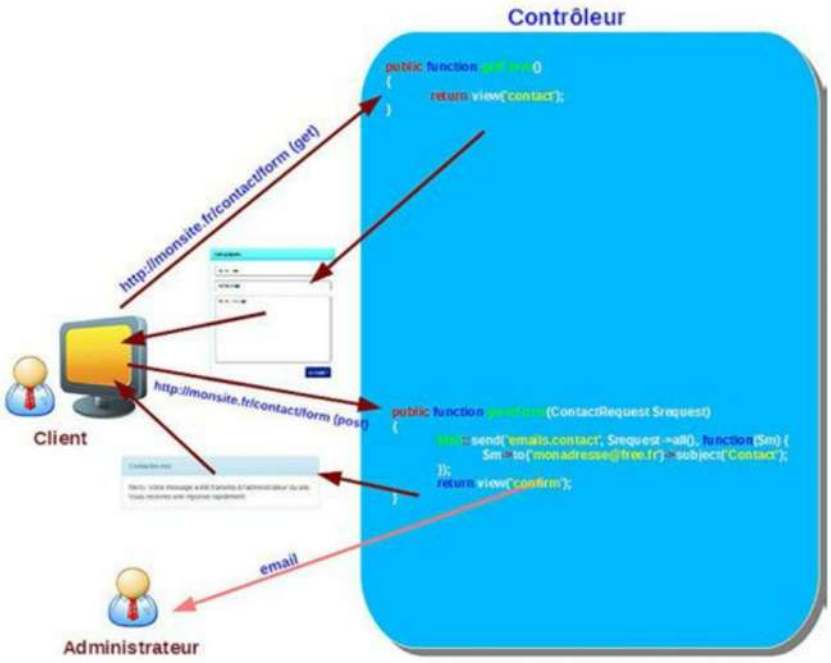
Prise de contact sur mon beau site

Réception d'une prise de contact avec les éléments suivants :

- Nom : Durant
- Email : durand@chezmoi.fr
- Message : Je trouve votre site très laid...

Le courriel pour l'administrateur

Voici une illustration globale du fonctionnement :



Le fonctionnement global

En résumé

- La validation est une étape essentielle de vérification des entrées du client.
- On dispose de nombreuses règles de validation.
- Le validateur génère des erreurs explicites à afficher au client.
- Pour obtenir les textes des erreurs en français, il faut aller chercher les traductions et les placer dans le bon dossier.
- Laravel permet l'envoi simple de courriels.

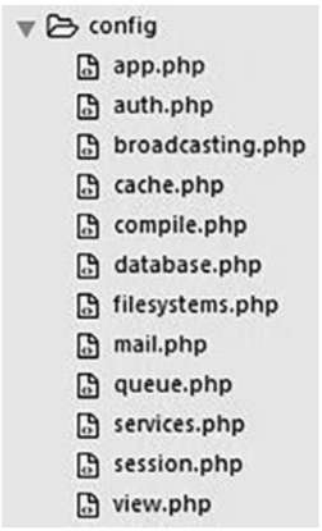
Copyright © 2016 Eyrolles.

8 Configuration et session

Dans ce chapitre, nous étudierons la configuration et la gestion des sessions avec un exemple simple d'envoi et d'enregistrement de fichiers images dans un dossier à partir d'un formulaire.

La configuration

Tout ce qui concerne la configuration de Laravel se trouve dans le dossier `config`.



Le dossier de configuration

Copyright © 2016 Eyrolles.

On a déjà eu l'occasion d'intervenir sur le fichier concernant les courriels. Les fichiers de configuration contiennent en fait juste un tableau avec des clés et des valeurs. Voici un exemple pour les vues :

```
<?php
'paths'=>[
  realpath(base_path('resources/views'))
],
```

Ici, la clé `'paths'` est associée à la valeur `realpath(base_path('resources/views'))`. Pour récupérer une valeur, il suffit d'utiliser sa clé avec la façade `Config` et la méthode `get` :

```
<?php
Config::get('view.paths');
```

On utilise le nom du fichier (`view`) et celui de la clé (`paths`) séparés par un point.

Il existe aussi un *helper* pour simplifier la syntaxe :

```
<?php
config('view.paths');
```

On peut de la même façon fixer une valeur :

```
<?php
Config::set('view.paths', base_path().'/mes_vues');
```

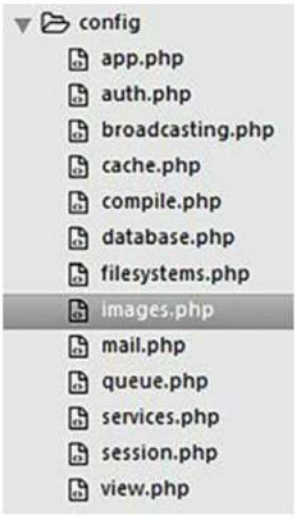
Si j'écris effectivement cela, mes vues, au lieu d'être cherchées dans le dossier `resources/views`, seront cherchées dans le dossier `mes_vues`.

On peut aussi fixer une valeur avec l'*helper* en passant un tableau comme paramètre :

```
<?php
config(['view.paths'=>base_path().'/mes_vues']);
```

Vous pouvez évidemment créer vos propres fichiers de configuration. Pour l'exemple de ce chapitre, on aura besoin justement d'utiliser une configuration. Comme notre application doit enregistrer des fichiers d'images, il faut définir l'emplacement et le nom du dossier de destination. On crée donc un fichier `images.php`.

Copyright © 2016 Eyrolles.

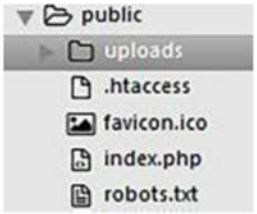


Le fichier de configuration des images

Dans ce fichier, on définit le chemin du dossier :

```
<?php
return ['path'=>'uploads'];
```

Et l'on crée aussi le dossier correspondant :



Le dossier pour les images

Les sessions

La façade `Session` de Laravel facilite leur gestion. Vous pouvez ainsi créer une variable de session :

```
<?php
Session::put('cle', 'valeur');

// Helper donnant le même résultat :
session(['cle'=>'valeur']);
```

Copyright © 2016 Eyrolles.

Vous pouvez aussi récupérer une valeur à partir de sa clé :

```
<?php
$valeur=Session::get('cle');

// Avec l'helper :
$valeur=session('cle');
```

Il est souvent utile (ce sera le cas pour notre exemple) de savoir si une certaine clé est présente en session :

```
<?php
if (Session::has('error'))

// Avec l'helper :
if (session()->has('error'))
```

Ces informations demeurent pour le même client à travers ses requêtes. On se contente d'indiquer un couple clé-valeur à Laravel et il s'occupe de tout.

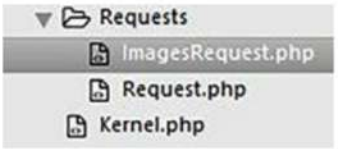
Ce ne sont là que les méthodes de base pour les sessions, utiles pour notre exemple. Vous trouverez tous les renseignements complémentaires dans la documentation : *https://laravel.com/docs/5.2/session*.

La requête de formulaire

Nous allons encore avoir besoin d'une requête de formulaire pour la validation. Comme nous l'avons déjà vu, nous utilisons la commande d'`Artisan` pour la créer :

```
php artisan make:request ImagesRequest
Request created successfully.
```

On trouve le fichier bien rangé.



La requête de formulaire pour les images

On complète le code comme suit :

```
<?php
namespace App\Http\Requests;

use App\Http\Requests\Request;
```

Copyright © 2016 Eyrolles.

```
class ImagesRequest extends Request {
  /**
   * Determine if the user is authorized to make this request.
   *
   * @return bool
   */
  public function authorize()
  {
    return true;
  }

  /**
   * Get the validation rules that apply to the request.
   *
   * @return array
   */
  public function rules()
  {
    return ['image'=>'required|image'];
  }
}
```

Nous définissons seulement deux règles pour le champ `image` :

- le champ est obligatoire (`required`) ;
- ce doit être une image (`image`).

Notre validation est prête.

Les routes et le contrôleur

On aura besoin de deux routes :

```
<?php
Route::get('photo', 'PhotoController@getForm');
Route::post('photo', 'PhotoController@postForm');
```

Notre contrôleur s'appelle `PhotoController` et voici son code :

```
<?php
namespace App\Http\Controllers;

use App\Http\Requests\ImagesRequest;

class PhotoController extends Controller
{
  public function getForm()
  {
    return view('photo');
  }
```

Copyright © 2016 Eyrolles.

```
  public function postForm(ImagesRequest $request)
  {
    $image=$request->file('image');
    if ($image->isValid())
    {
      $chemin=config('images.path');
      $extension=$image->getClientOriginalExtension();
      do {
        $nom=str_random(10) . '.' . $extension;
      } while(file_exists($chemin . '/' . $nom));
      if ($image->move($chemin, $nom)) {
        return view('photo_ok');
      }
    }
    return redirect('photo')
      ->with('error','Désolé mais votre image ne peut pas être envoyée !');
  }
}
```

Donc nous aurons une URL *http://monsite.fr/photo* :

- avec le verbe `get` pour la demande du formulaire ;
- avec le verbe `post` pour la soumission du formulaire et l'envoi du fichier image associé.

En ce qui concerne le traitement de la soumission, vous remarquez qu'on récupère le chemin du dossier d'enregistrement prévu dans la configuration :

```
<?php
$chemin=config('images.path');
```

Pour récupérer le fichier envoyé, j'ai utilisé la requête et la méthode `file` :

```
<?php
$image=$request->file('image');
```

Pour récupérer l'extension originelle, on utilise la méthode `getClientOriginalExtension` qui est une des méthodes de `Symfony\Component\HttpFoundation\File`, comme la méthode `isValid` qui vérifie la validité du fichier.

On génère un nom aléatoire avec l'*helper* `str_random` en définissant 10 caractères et on vérifie que le nom n'est pas déjà pris.

Enfin, on enregistre l'image avec la méthode `move`. Si tout se passe bien, on retourne la vue `photo_ok`. Sinon, on redirige (`redirect`) vers l'URL `photo` en prévoyant dans la session (`with`) une variable `error` avec la valeur `'Désolé mais votre image ne peut pas être envoyée !'`.

Copyright © 2016 Eyrolles.

Les vues

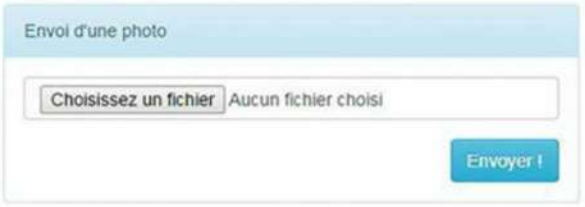
On utilisera le template des chapitres précédents (`resources/views/template.blade.php`) :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Mon joli site</title>
    {!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/
bootstrap.min.css') !!}
    {!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/
bootstrap-theme.min.css') !!}
    <!--[if lt IE 9]>
      {{ Html::style('https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.
js') }}
      {{ Html::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.
min.js') }}
    <![endif]-->
    <style> textarea {resize:none;} </style>
  </head>
  <body>
    @yield('contenu')
  </body>
</html>
```

Voici la vue pour le formulaire (`resources/views/photo.blade.php`) :

```
@extends('template')

@section('contenu')
  <br>
  <div class="col-sm-offset-4 col-sm-4">
    <div class="panel panel-info">
      <div class="panel-heading">Envoi d'une photo</div>
      <div class="panel-body">
        @if(session()->has('error'))
          <div class="alert alert-danger">{!! session('error') !!}</div>
        @endif
        {!! Form::open(['url'=>'photo', 'files'=>true]) !!}
          <div class="form-group {!! $errors->has('image') ? 'has-
error' : '' !!}">
            {!! Form::file('image', ['class'=>'form-control']) !!}
            {!! $errors->first('image', '<small class="help-block">:message</
small>') !!}
          </div>
          {!! Form::submit('Envoyer !', ['class'=>'btn btn-info pull-
right']) !!}
        {!! Form::close() !!}
      </div>
    </div>
  </div>
@endsection
```

Copyright © 2016 Eyrolles.



Le formulaire

Cette vue fait apparaître une possibilité de `Blade` qu'on n'avait pas encore rencontrée, celle d'utiliser des conditions :

```
@if(session()->has('error'))
  <div class="alert alert-danger">{{session('error')}}</div>
@endif
```

Ici, on teste avec `@if` la présence de la clé `'error'` dans la session. Si cette clé est présente, alors on fait apparaître une barre d'alerte avec le texte contenu en session, récupéré avec l'*helper* `session`.

Remarquez aussi comment est créé le formulaire :

```
Form::open(['url'=>'photo', 'files'=>true])
```

Le fait d'ajouter l'attribut `files` avec la valeur `true` a pour effet de faire apparaître le type *mime* nécessaire pour associer un fichier lors de la soumission :

```
enctype="multipart/form-data"
```

En cas d'erreur de validation, le message est affiché et la bordure du champ devient rouge.

Erreur de validation

En cas de problème dans la mémorisation du fichier, on retourne un message par l'intermédiaire de la session et on l'affiche dans une barre d'alerte :

Copyright © 2016 Eyrolles.

Erreur de mémorisation du fichier

Et voici la vue pour la confirmation en retour (`app/views/photo_ok.blade.php`) :

```
@extends('template')

@section('contenu')
  <br>
  <div class="col-sm-offset-3 col-sm-6">
    <div class="panel panel-info">
      <div class="panel-heading">Envoi d'une photo</div>
      <div class="panel-body">
        Merci. Votre photo a bien été reçue et enregistrée.
      </div>
    </div>
  </div>
@endsection
```

La vue de confirmation

On retrouve normalement le fichier bien rangé dans le dossier prévu.

Le fichier dans le dossier prévu

Voici le schéma de fonctionnement.

Copyright © 2016 Eyrolles.

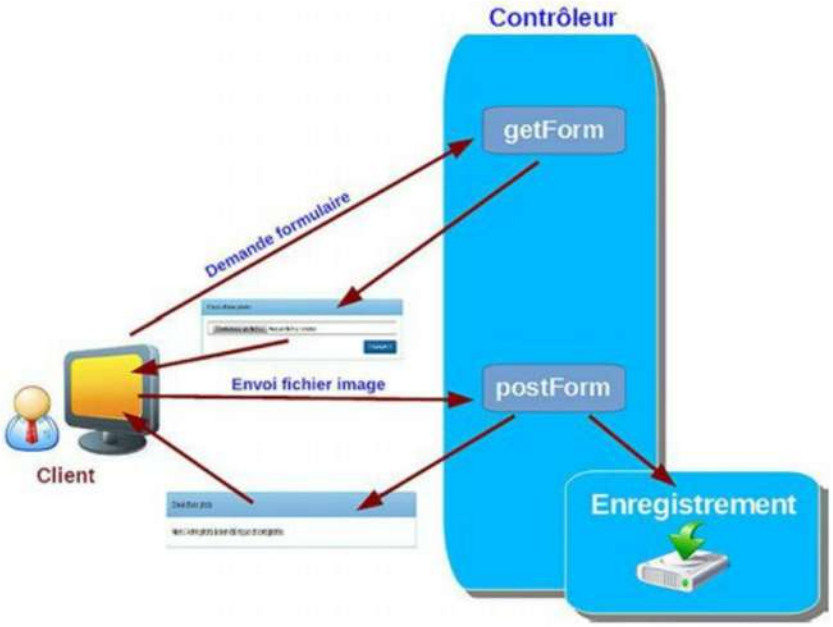


Schéma de fonctionnement

En résumé

- Les fichiers de configuration permettent de mémoriser facilement des ensembles clé-valeur et sont gérés par la façade `Config` ou l'*helper* `config`.
- Les sessions servent à mémoriser des informations concernant un client et sont facilement manipulables avec la façade `Session` ou avec l'*helper* `session`.

Copyright © 2016 Eyrolles.

9 L'injection de dépendances

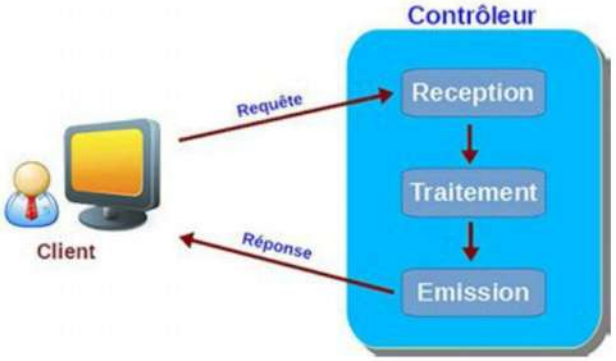
Dans ce chapitre, nous allons reprendre l'exemple précédent de l'envoi de photos en nous posant des questions d'organisation du code. Laravel n'est pas seulement un framework pratique, c'est aussi un style de programmation. Il vaut mieux évoquer ce style le plus tôt possible dans l'apprentissage pour prendre rapidement les bonnes habitudes.

Vous pouvez très bien créer un site complet dans le fichier des routes ; vous pouvez aussi vous contenter de contrôleurs pour effectuer tous les traitements nécessaires. Je vous propose une autre approche, plus en accord avec ce que nous offre Laravel.

Le problème et sa solution

Problème

Un contrôleur a pour mission de réceptionner les requêtes et d'envoyer les réponses. Entre les deux, il y a évidemment du traitement à effectuer pour construire la réponse ; parfois c'est très simple, parfois plus long et délicat. Globalement, nous avons pour un contrôleur le fonctionnement suivant.



Fonctionnement d'un contrôleur

Copyright © 2016 Eyrolles.

Reprenons la méthode `postForm` de notre contrôleur `PhotoController` du précédent chapitre :

```
<?php
public function postForm(ImagesRequest $request)
{
  $image=$request->file('image');
  if ($image->isValid())
  {
    $chemin=config('images.path');
    $extension=$image->getClientOriginalExtension();
    do {
      $nom=str_random(10) . '.' . $extension;
    } while (file_exists($chemin . '/' . $nom));
    if ($image->move($chemin, $nom)) {
      return view('photo_ok');
    }
  }
  return redirect('photo')
    ->with('error','Désolé mais votre image ne peut pas être envoyée !');
}
```

Qu'avons-nous comme traitement ? On récupère les références de l'image transmise, on récupère le dossier de destination dans la configuration, on trouve l'extension du fichier image, on génère un nom et enfin on enregistre l'image.

La question est : est-ce qu'un contrôleur doit savoir comment s'effectue ce traitement ? Si vous avez plusieurs contrôleurs dans votre application qui doivent effectuer le même traitement, vous allez multiplier cette mise en place. Imaginez que vous ayez ensuite envie de modifier l'enregistrement des images, par exemple en les mettant dans une base de données, vous allez devoir retoucher le code de tous vos contrôleurs ! La répétition de code n'est jamais une bonne chose. Une saine règle de programmation veut qu'on commence à se poser des questions sur l'organisation du code dès qu'on fait des copies.

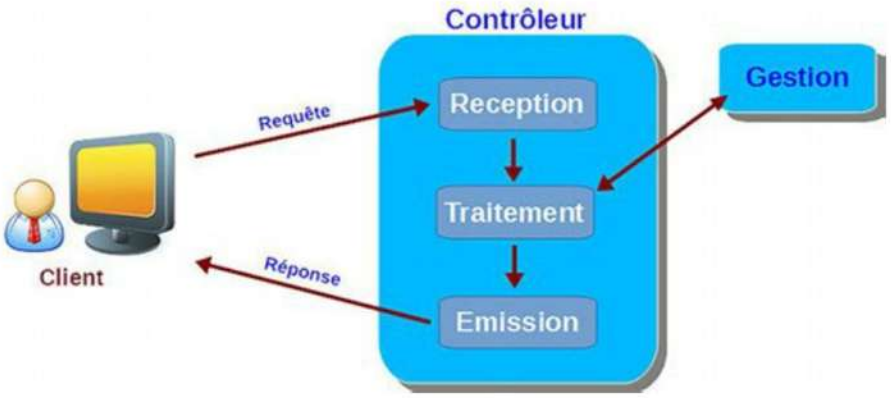
Cette préconisation est connue sous l'acronyme DRY *(Don't Repeat Yourself)*.

Un autre élément à prendre en compte est la testabilité des classes. Nous verrons cet aspect important du développement trop souvent négligé. Pour qu'une classe soit testable, il faut que sa mission soit simple et parfaitement identifiée et il ne faut pas qu'elle soit étroitement liée avec une autre classe. En effet, cette dépendance rend les tests plus difficiles.

Solution

Alors quelle est la solution ? L'injection de dépendances ! Voyons de quoi il s'agit.

Copyright © 2016 Eyrolles.



L'injection de la gestion

Une nouvelle classe entre en jeu pour la gestion ; c'est elle qui est effectivement chargée du traitement, le contrôleur faisant juste appel à ses méthodes. Comment cette classe est-elle injectée dans le contrôleur ? Voici le code du contrôleur modifié :

```
<?php
namespace App\Http\Controllers;

use App\Http\Requests\ImagesRequest;
use App\Gestion\PhotoGestion;

class PhotoController extends Controller {
  public function getForm()
  {
    return view('photo');
  }


  public function postForm(
    ImagesRequest $request,
    PhotoGestion $photogestion)
  {
    if ($photogestion->save($request->file('image'))) {
      return view('photo_ok');
    }
    return redirect('photo')
      ->with('error','Désolé mais votre image ne peut pas être envoyée !');
  }
```

Vous remarquez qu'au niveau de la méthode `postForm`, il y a un nouveau paramètre de type `App\Gestion\PhotoGestion`. On utilise la méthode `save` de la classe ainsi injectée pour réaliser le traitement.

De cette façon, le contrôleur ignore totalement comment se fait la gestion ; il sait juste que la classe `PhotoGestion` sait s'en occuper. Il se contente d'utiliser la méthode de cette classe qui est « injectée ». Maintenant, vous vous demandez sans doute comment cette classe est injectée là, en d'autres termes comment et où est créée cette instance. Eh bien Laravel est assez malin pour le faire lui-même.

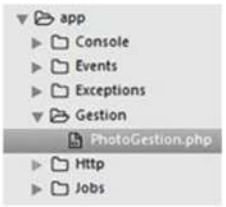
Copyright © 2016 Eyrolles.

PHP est très tolérant sur les types des variables. Lorsque vous en déclarez une, vous n'êtes pas obligé de préciser que c'est un `string` ou un `array`. PHP devine le type selon la valeur affectée. Il en est de même pour les paramètres des fonctions. Cependant, personne ne vous empêche de déclarer un type comme je l'ai fait ici pour le paramètre de la méthode (pour le moment malheureusement, PHP ne reconnaît que les tableaux et les classes). C'est même indispensable pour que Laravel sache quelle classe est concernée. Étant donné que je déclare le type, Laravel est capable de créer une instance de ce type et de l'injecter dans le contrôleur.

Pour trouver la classe, Laravel utilise l'introspection (reflection en anglais) de PHP, qui permet d'inspecter le code en cours d'exécution. Elle aide aussi à manipuler du code et donc à créer par exemple un objet d'une certaine classe. Vous trouverez tous les renseignements dans le manuel PHP : *http://php.net/manual/fr/book.reflection.php*.

La gestion

Maintenant qu'on a dit au contrôleur qu'une classe s'occupe de la gestion, il nous faut la créer. Pour bien organiser notre application, on crée un nouveau dossier et on place notre classe dedans.



Le dossier de gestion

Le codage ne pose aucun problème parce qu'il est identique à ce qu'on avait dans le contrôleur :

```
<?php
namespace App\Gestion;

class PhotoGestion
{
  public function save($image)
  {
    if ($image->isValid())
    {
      $chemin=config('images.path');
      $extension=$image->getClientOriginalExtension();
      do {
        $nom=str_random(10) . '.' . $extension;
      } while (file_exists($chemin . '/' . $nom));
      return $image->move($chemin, $nom);
    }
    return false;
  }
}
```

Copyright © 2016 Eyrolles.

N'oubliez pas les espaces de noms !

Notre code est alors parfaitement organisé et facile à maintenir et à tester.
Allons un peu plus loin : créons une interface pour notre classe.

L'interface pour la classe PhotoGestion

```
<?php
namespace App\Gestion;

interface PhotoGestionInterface
{
  public function save($image);
}
```

Il suffit ensuite d'en informer la classe `PhotoGestion` :

```
<?php
class PhotoGestion implements PhotoGestionInterface
```

Il serait bien maintenant de référencer l'interface dans notre contrôleur :

```
<?php
namespace App\Http\Controllers;

use App\Http\Requests\ImagesRequest;
use App\Gestion\PhotoGestionInterface;

class PhotoController extends Controller
{
  public function getForm()
  {
    return view('photo');
  }

  public function postForm(
    ImagesRequest $request,
    PhotoGestionInterface $photogestion)
  {
   if ($photogestion->save($request->file('image'))) {
```

Copyright © 2016 Eyrolles.

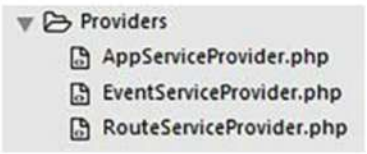
```
        return view('photo_ok');
    }
    return redirect('photo')
      ->with('error','Désolé mais votre image ne peut pas être envoyée !');
  }
```

Le souci, c'est que Laravel n'arrive pas à deviner la classe à instancier à partir de cette interface.

Laravel ne sait pas instancier l'interface.

Comment s'en sortir ?

Lorsque j'ai présenté la structure de Laravel, j'ai mentionné la présence de fournisseurs.



Les fournisseurs

Un fournisseur sert tout simplement à procéder à des initialisations (événements, *middlewares*) et surtout des liaisons de dépendances. Laravel possède un conteneur de dépendances qui constitue le cœur de son fonctionnement. C'est grâce à ce conteneur qu'on établira une liaison entre une interface et une classe.

Ouvrez le fichier `app\Providers\AppServiceProvider.php` et ajoutez le code suivant :

```
<?php
public function register()
{
  ...
  $this->app->bind(
    'App\Gestion\PhotoGestionInterface',
    'App\Gestion\PhotoGestion'
  );
}
```

Copyright © 2016 Eyrolles.

La méthode `register` est activée au démarrage de l'application ; c'est l'endroit idéal pour notre liaison. Ici, on dit à l'application (`app`) d'établir une liaison (`bind`) entre l'interface `App\Gestion\PhotoGestionInterface` et la classe `App\Gestion\PhotoGestion`. Ainsi, chaque fois qu'on se référera à cette interface dans une injection, Laravel saura quelle classe instancier. Si on veut changer la classe de gestion, il suffit de modifier le code du fournisseur.

Maintenant notre application fonctionne.

Si vous obtenez un message d'erreur vous disant que l'interface ne peut pas être instanciée lancez la commande `php artisan clear-compiled`.

En résumé

- Un contrôleur doit déléguer toute tâche qui ne relève pas de sa compétence.
- L'injection de dépendances permet de bien séparer les tâches, de simplifier la maintenance du code et les tests unitaires.
- Les fournisseurs réalisent les initialisations, en particulier les liaisons de dépendances entre interfaces et classes.

Copyright © 2016 Eyrolles.

Copyright © 2016 Eyro les.

Deuxième partie

Les bases de données

Lorsqu'on veut créer un site dynamique, une base de données est nécessaire ; il faut donc des outils pour la gérer. Ceux proposés par Laravel sont très efficaces. Il possède un système de migration et de population, un gestionnaire de requêtes SQL et surtout le mapping objet-relationnel `Eloquent` qui simplifie de façon drastique la gestion des données.

Laravel supporte quatre systèmes de bases de données : MySQL, PostgreSQL, SQLite et SQL Server. Quel que soit le système utilisé, le codage reste le même et il suffit de régler la configuration.

Dans cette partie, nous verrons aussi le système d'authentification de Laravel qui prend en charge les fonctionnalités essentielles : connexion, déconnexion et récupération du mot de passe.

Enfin, on explorera un peu mieux `Artisan` en créant une commande personnalisée.

Copyright © 2016 Eyrolles.

Copyright © 2016 Eyro les.

10 Migrations et modèles

Dans ce chapitre, nous allons aborder les bases de données. C'est un vaste sujet auquel Laravel apporte des réponses efficaces. Nous allons commencer par étudier les migrations et les modèles.

Pour ce chapitre, je vais encore prendre un exemple simple en imaginant un formulaire pour s'inscrire à une lettre d'information. On se contentera d'envoyer une adresse électronique et de la mémoriser dans une table d'une base de données.

> À partir de ce chapitre, il serait souhaitable que vous installiez une barre de débogage. La plus utile est celle proposée par *barryvdh* : *https://github.com/barryvdh/laravel-debugbar*. Suivez les indications fournies pour l'installation, cela vous fera un bon exercice.

Copyright © 2016 Eyrolles.

Les migrations

Une migration permet de créer et de mettre à jour un schéma de base de données. Autrement dit, vous pouvez créer des tables, des colonnes dans ces tables, en supprimer, créer des index… Tout ce qui concerne la maintenance de vos tables peut être pris en charge par cet outil.

Configuration de la base

Vous devez dans un premier temps avoir une base de données. Laravel sait gérer les bases de type MySQL, PostgreSQL, SQLite et SQL Server. Je ferai tous les exemples avec MySQL, mais tout le code sera aussi valable pour les autres types de bases.

Il faut indiquer où se trouve votre base, son nom, le nom de l'utilisateur, le mot de passe dans le fichier de configuration `.env` :

```
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret
```

Ici, nous avons les valeurs par défaut à l'installation de Laravel.

Voici par exemple mes réglages pour ma base de test nommée `tuto` avec MySQL en local non sécurisé :

```
DB_DATABASE=tuto
DB_USERNAME=root
DB_PASSWORD=
```

Artisan

Nous avons déjà utilisé cet outil en ligne de commande. Vous obtenez un aperçu de toutes les commandes qu'il propose en entrant :

```
php artisan
```

La liste est longue. Pour ce chapitre, nous allons nous intéresser uniquement à celles qui concernent les migrations :

```
make
  make:migration      Create a new migration file
migrate
  migrate:install     Create the migration repository
  migrate:refresh     Reset and re-run all migrations
  migrate:reset       Rollback all database migrations
  migrate:rollback    Rollback the last database migration
  migrate:status      Show the status of each migration
```

Installer la migration

On commence par installer la migration :

```
php artisan migrate:install
Migration table created successfully.
```

Si vous regardez l'effet dans votre base, vous allez voir qu'une table a été créée.

	Table	Action	
☐	migrations	Afficher	Structure
	1 table	Somme	

La table des migrations

Copyright © 2016 Eyrolles.

C'est dans cette table que seront mémorisées toutes vos actions au niveau du schéma de la base.

Créer la migration

La deuxième étape consiste à créer la migration pour notre table :

```
php artisan make:migration create_emails_table
Created Migration: 2015_12_27_210631_create_emails_table
```

Si vous regardez dans le dossier `database/migrations`, vous trouvez un fichier ayant cet aspect : `2015_12_27_210631_create_emails_table.php` (la partie numérique qui inclut la date sera évidemment différente pour vous).

La migration créée

Il y a déjà des migrations présentes ; à quoi servent-elles ?

Il y a déjà effectivement deux migrations présentes :

- `table users` : c'est une migration de base pour créer une table des utilisateurs ;
- `table password_resets` : c'est une migration liée à la précédente qui permet de gérer le renouvellement des mots de passe en toute sécurité.

Nous nous intéresserons à ces migrations lorsque nous étudierons l'authentification dans un chapitre ultérieur. Comme nous n'allons pas en avoir immédiatement besoin, le mieux est de les supprimer pour le moment pour éviter de créer des tables inutiles.

Voici le contenu de la migration que nous venons de créer :

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateEmailsTable extends Migration {
  /**
   * Run the migrations.
   *
   * @return void
  * @return void
   */
```

Copyright © 2016 Eyrolles.

```
    public function up()
    {
        //
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        //
    }
}
```

On dispose dans cette classe de deux fonctions :

- `up` : code de création ;
- `down` : code de suppression.

On veut créer une table `emails` avec un identifiant auto-incrémenté et un champ `email` de type texte et de longueur 100. Voilà le code correspondant :

```
<?php
public function up()
{
    Schema::create('emails', function(Blueprint $table) {
        $table->increments('id');
        $table->string('email', 100);
    });
}
```

On demande au constructeur de schéma (`Schema`) de créer (`create`) la table `emails`. Dans la fonction anonyme on définit les deux colonnes qu'on veut pour la table.

Pour la méthode `down`, on supprime juste la table avec un `drop` :

```
<?php
public function down()
{
    Schema::drop('emails');
}
```

Notre migration est créée.

Utiliser la migration

Lançons la migration (utilisation de sa méthode `up`) :

```
php artisan migrate
Migrated: 2015_12_27_210631_create_emails_table
```


Copyright © 2016 Eyrolles.

Si on regarde dans la base, on trouve la table `emails` avec ses deux colonnes.

#	Nom	Type	Interclassement	Attributs	Null	Défaut	Extra
1	id	int(10)		UNSIGNED	Non	*Aucune*	AUTO_INCREMENT
2	email	varchar(100)	utf8_unicode_ci		Non	*Aucune*	

La table emails

Si vous avez fait une erreur, vous pouvez revenir en arrière avec un `rollback` qui annule la dernière migration effectuée (utilisation de la méthode `down` de la migration) :

```
php artisan migrate:rollback
Rolled back: 2015_12_27_210631_create_emails_table
```

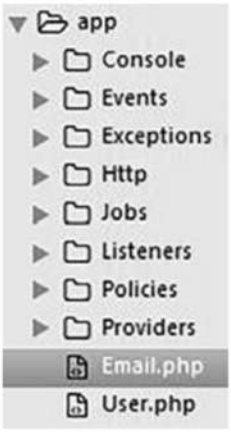
La table a été supprimée de la base. Comme on aura besoin de cette table, on relance la migration. On peut aussi effectuer un rafraîchissement de toutes les migrations avec la commande `refresh` (`rollback` puis nouveau lancement de toutes les migrations).

Eloquent, un ORM très performant

Laravel propose `Eloquent`, un ORM (*Object-Relational Mapping* ou mappage objet-relationnel) très performant : tous les éléments de la base de données ont une représentation sous forme d'objets manipulables. L'intérêt est de simplifier grandement les opérations sur la base, comme nous allons le voir dans toute cette partie de l'ouvrage.

Avec `Eloquent`, une table est représentée par une classe qui étend la classe `Model`. Pour notre table `emails`, on utilisera à nouveau `Artisan` pour la création du modèle :

```
php artisan make:model Email
```



Le nouveau modèle pour les adresses électroniques

En voici la trame de base :

Copyright © 2016 Eyrolles.

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Email extends Model
{
  //
}
```

On complétera le code comme suit :

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Email extends Model
{
  protected $table='emails';
  public $timestamps=false;
}
```

Vous voyez, c'est tout simple ! On renseigne le nom de la table associée au modèle. De plus, `Eloquent` tient par défaut à jour des colonnes `created_at` et `updated_at` dans la table. Comme nous ne les avons pas prévues, nous désactivons cette action en indiquant `false` pour la propriété `timestamps`. Par souci de simplicité pour le moment, on mettra ce modèle directement dans le dossier `app`, mais il est évident que pour une application réelle, on organisera les dossiers pour bien ranger tous les fichiers.

Si on ne renseigne pas la propriété `$table`, Laravel déduit le nom de la table à partir du nom du modèle. Autrement dit dans notre cas, on pourrait éviter cette ligne de code.

Nous allons voir comment utiliser cette classe en construisant notre petite application.

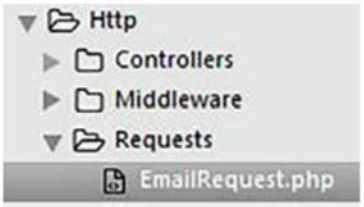
Validation

Pour la validation, on créera encore une requête de formulaire :

```
php artisan make:request EmailRequest
Request created successfully.
```

On trouve la requête dans son dossier.

Copyright © 2016 Eyrolles.



La requête de formulaire

La voici avec le code complété :

```
<?php
namespace App\Http\Requests;

use App\Http\Requests\Request;

class EmailRequest extends Request
{
  /**
   * Determine if the user is authorized to make this request.
   *
   * @return bool
   */
  public function authorize()
  {
    return true;
  }

  /**
   * Get the validation rules that apply to the request.
   *
   * @return array
   */
  public function rules()
  {
    return ['email'=>'required|email|unique:emails'];
  }
}
```

On a trois règles :

- `required` : le champ est requis ;
- `email` : l'adresse électronique doit être valide ;
- `unique` : l'adresse ne doit pas déjà exister (`unique`) dans la table `emails` (on sous-entend qu'il s'agit de la colonne `email`).

Remarquez la puissance de la troisième règle : `Eloquent` vérifiera que notre adresse n'existe pas déjà dans la table !

Copyright © 2016 Eyrolles.

Routes

Il nous faut deux routes :

```
<?php
Route::get('email', 'EmailController@getForm');
Route::post('email', ['uses'=>'EmailController@postForm',
'as'=>'storeEmail']);
```

Remarquez que la seconde route est nommée (`storeEmail`).

L'URL de base sera donc : *http://monsite.fr/email*.

Contrôleur

Créons le contrôleur `EmailController`.

Le contrôleur EmailController

Le code du contrôleur reprend l'essentiel de ce que nous avons vu dans les chapitres précédents en utilisant à nouveau la validation injectée :

```
<?php
namespace App\Http\Controllers;

use App\Email;
use App\Http\Requests\EmailRequest;

class EmailController extends Controller
{
  public function getForm()
  {
    return view('email');
  }

  public function postForm(EmailRequest $request)
  {
    $email=new Email;
    $email->email=$request->input('email');
    $email->save();
    return view('email_ok');
  }
}
```

Copyright © 2016 Eyrolles.

La nouveauté réside uniquement dans l'utilisation du modèle :

```
<?php
$email=new Email;
$email->email=$request->input('email');
$email->save();
```

Ici, on crée une nouvelle instance de `Email`. On affecte l'attribut `email` avec la valeur de l'entrée. Enfin, on demande au modèle d'enregistrer cette ligne effectivement dans la table (`save`).

Vues

Utilisons le même template que dans les précédents chapitres. Voici la vue pour le formulaire (`resources/views/email.blade.php`) :

```
@extends('template')

@section('contenu')
  <br>
  <div class="col-sm-offset-4 col-sm-4">
    <div class="panel panel-info">
      <div class="panel-heading">Inscription à la lettre d'information</div>
      <div class="panel-body">
        {!! Form::open(['route'=>'storeEmail']) !!}
          <div class="form-group {!! $errors->has('email') ? 'has-
error' : '' !!}">
            {!! Form::email('email', null, array('class'=>'form-control',
'placeholder'=>'Entrez votre email')) !!}
            {!! $errors->first('email', '<small class="help-block">:message</
small>') !!}
          </div>
          {!! Form::submit('Envoyer !', ['class'=>'btn btn-info pull-
right']) !!}
        {!! Form::close() !!}
      </div>
    </div>
  </div>
@endsection
```

Cette vue ne présente aucune nouveauté pour vous, si ce n'est l'utilisation du nom de la route ; elle répond à l'URL (avec le verbe `get`) *http://monsite.fr/email*.

L'aspect est le suivant.

Copyright © 2016 Eyrolles.

Le formulaire

Voici la vue de confirmation (`resources/views/email_ok.blade.php`) :

```
@extends('template')

@section('contenu')
  <br>
  <div class="col-sm-offset-3 col-sm-6">
    <div class="panel panel-info">
      <div class="panel-heading">Inscription à la lettre d'information</div>
      <div class="panel-body">
        Merci. Votre adresse a bien été prise en compte.
      </div>
    </div>
  </div>
@endsection
```

Elle présente l'aspect suivant.

La confirmation

Fonctionnement

Voyons si tout se passe bien.

1. Je soumets une adresse.

Soumission d'une adresse

Copyright © 2016 Eyrolles.

2. Je reçois la confirmation.

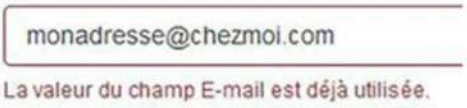
La confirmation

3. Je regarde dans la base.

id	email
1	monadresse@chezmoi.com

L'adresse dans la base

4. Je soumets la même adresse.



Soumission d'une adresse existante

Voyons un peu les requêtes construites par `Eloquent` avec par exemple la soumission de l'adresse *toto@gui.com* (vous les trouvez à la rubrique ***Queries*** de la barre de débogage) :

```
select count(*) as aggregate from 'emails' where 'email'='toto@gui.com'
insert into 'emails' ('email') values ('toto@gui.com')
```

La première requête est destinée à tester la présence éventuelle de l'adresse dans la table pour répondre à la règle `unique`. La seconde insère l'enregistrement dans la table. Vous voyez qu'`Eloquent` vous simplifie la tâche : vous n'avez pas besoin d'écrire les requêtes SQL, il le fait pour vous. Vous vous contentez de manipuler un objet.

N'hésitez pas à regarder les informations de la barre de débogage, vous y trouverez de précieux renseignements sur les requêtes (HTTP et SQL), les vues utilisées, les routes, les délais, les exceptions levées… Vous obtenez aussi un historique en cliquant sur la petite image de dossier.

Ouvrir l'historique de la barre

Copyright © 2016 Eyrolles.

L'organisation du code

Posons-nous à nouveau la question de l'organisation du code. Dans le contrôleur, nous avons écrit la gestion du modèle :

```
<?php
$email=new Email;
$email->email=$request->input('email');
$email->save();
```

Autrement dit, nous avons lié de façon étroite le contrôleur et le modèle. Supposons que nous apportions des modifications dans notre base de données et que nous placions l'adresse électronique dans une autre table. Nous devrions évidemment intervenir dans le code du contrôleur pour tenir compte de cette modification.

Vous pouvez évidemment considérer que c'est peu probable, que la modification du code n'est pas très importante… Pourtant, ici on a une application très simple, alors que dans une situation réelle, les modèles sont nombreux et la question devient plus pertinente.

Première version

Dans cet ouvrage, je m'efforce de vous entraîner à prendre de bonnes habitudes. Plutôt que d'instancier directement une classe dans une autre, il vaut mieux une injection et laisser faire le conteneur. Regardez cette nouvelle version du contrôleur :

```
<?php
namespace App\Http\Controllers;

use App\Email;
use App\Http\Requests\EmailRequest;

class EmailController extends Controller
{
  public function getForm()
  {
    return view('email');
  }

  public function postForm(
    EmailRequest $request,
    Email $email)
  {
    $email->email=$request->input('email');
    $email->save();
    return view('email_ok');
  }
}
```

Copyright © 2016 Eyrolles.

Le modèle est désormais injecté dans la méthode ; c'est plus élégant et efficace. Si jamais vous modifiez le modèle, vous n'avez plus qu'un changement de code limité sur le contrôleur. Toutefois, ce n'est pas encore parfait.

Deuxième version

Dans l'idéal, on veut que notre contrôleur ne soit pas du tout concerné par un changement dans la gestion des modèles. Voici une façon de procéder :

```
<?php
namespace App\Http\Controllers;

use App\Http\Requests\EmailRequest;
use App\Repositories\EmailRepository;

class EmailController extends Controller
{
  public function getForm()
  {
    return view('email');
  }

  public function postForm(
    EmailRequest $request,
    EmailRepository $emailRepository)
  {
    $emailRepository->save($request->input('email'));
    return view('email_ok');
  }
}
```

J'injecte une classe de gestion qui possède la méthode `save`. Voici le contrat avec une interface (`app/Repositories/EmailRepositoryInterface`) :

```
<?php
namespace App\Repositories;

interface EmailRepositoryInterface
{
  public function save($mail);
}
```

Et voici la classe qui implémente cette interface (`app/Repositories/EmailRepository`) :

```
<?php
namespace App\Repositories;

use App\Email;
```

Copyright © 2016 Eyrolles.

```
class EmailRepository implements EmailRepositoryInterface
{
  protected $email;

  public function __construct(Email $email)
  {
    $this->email=$email;
  }

  public function save($mail)
  {
    $email=new $this->email;
    $email->email=$mail;
    $email->save();
  }
}
```

Le modèle est injecté dans cette classe. Je l'ai injecté dans le constructeur pour généraliser la démarche en imaginant qu'on créera d'autres méthodes qu'on peut regrouper ici pour gérer les enregistrements. Le code est maintenant parfaitement organisé, facile à modifier et à tester.

La gestion

Le *Design Pattern* `Repository` est un des plus répandus. Il permet de gérer la persistance des informations.

Troisième version

On peut enfin, comme on l'a déjà vu, référencer l'interface plutôt que la classe, mais il faut informer le conteneur de la dépendance. Modifiez comme suit le fichier `app/HTTP/Providers/AppServiceProvider.php` :

```
<?php
public function register()
{
  ...
  $this->app->bind(
    'App\Repositories\EmailRepositoryInterface',
    'App\Repositories\EmailRepository'
  );
}
```

Copyright © 2016 Eyrolles.

Et voici finalement le contrôleur :

```
<?php
namespace App\Http\Controllers;

use App\Http\Requests\EmailRequest;
use App\Repositories\EmailRepositoryInterface;

class EmailController extends Controller
{
  public function getForm()
  {
    return view('email');
  }

  public function postForm(
    EmailRequest $request,
    EmailRepositoryInterface $emailRepository)
  {
    $emailRepository->save($request->input('email'));
    return view('email_ok');
  }
}
```

Vu que le conteneur sait quelle classe instancier à partir de l'interface passée en paramètre, vous avez un code propre et facile à maintenir et à tester. Si vous changez d'avis sur la manière de stocker les adresses électroniques, il vous suffit de décider d'instancier une autre classe à partir de l'interface, tant que le contrat passé avec le contrôleur ne change pas !

En résumé

- La base de données doit être configurée pour fonctionner avec Laravel.
- Les migrations servent à intervenir sur le schéma des tables de la base.
- `Eloquent` permet une représentation des tables sous forme d'objets pour simplifier les manipulations des enregistrements.
- Il est judicieux de prévoir la gestion du modèle dans une classe injectée dans le contrôleur.
- La barre de débogage donne de précieux renseignements sur les requêtes.

Copyright © 2016 Eyrolles.

Copyright © 2016 Eyro les.

11 Les ressources

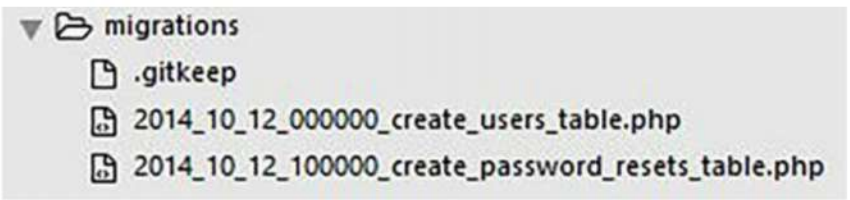
Dans ce chapitre, nous allons commencer à étudier les ressources qui permettent de créer un contrôleur *RESTful* adapté à une ressource. Comme exemple pratique, nous allons prendre le cas d'une table d'utilisateurs, une situation qui se retrouve dans la plupart des applications.

Les données

À partir d'un Laravel vierge, on commence par configurer la base comme on l'a vu au chapitre précédent et par installer la migration pour créer la table des migrations dans la base :

```
php artisan migrate:install
Migration table created successfully.
```

Lorsqu'on installe Laravel, on se retrouve avec deux migrations déjà présentes.



Les migrations présentes à l'installation

Pour le moment, on garde seulement la première, qui concerne la création de la table des utilisateurs. On va juste un peu la modifier :

Copyright © 2016 Eyrolles.

```php
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateUsersTable extends Migration
{
  /**
   * Run the migrations.
   *
   * @return void
   */
  public function up()
  {
    Schema::create('users', function(Blueprint $table)
    {
      $table->increments('id');
      $table->string('name')->unique();
      $table->string('email')->unique();
      $table->string('password', 60);
      $table->boolean('admin')->default(false);
      $table->rememberToken();
      $table->timestamps();
    });
  }

  /**
   * Reverse the migrations.
   *
   * @return void
   */
  public function down()
  {
    Schema::drop('users');
  }
}
```

On a les champs suivants :

- `id` : entier auto-incrémenté qui sera la clé primaire de la table ;
- `name` : texte (`varchar 255`) pour le nom (`unique`) ;
- `email` : texte (`varchar 255`) pour l'adresse électronique (`unique`) ;
- `password` : texte de 60 caractères pour le mot de passe (ce qui ne signifie pas que le mot de passe pourra avoir 60 caractères, parce qu'on en stocke ici une version cryptée) ;
- `admin` : valeur booléeenne pour indiquer s'il s'agit d'un administrateur avec `false` comme valeur par défaut ;
- `remember_token` : texte de 100 caractères qui servira pour l'authentification dans un chapitre ultérieur ;
- `created_at` et `updated_at` créés par la méthode `timestamps`.

Ensuite, on lance la migration :

Copyright © 2016 Eyrolles.

```
php artisan migrate
Migrated: 2015_12_27_210631_create_users_table
```

La table est créée.

#	Nom	Type	Interclassement	Attributs	Null	Défaut	Extra
1	id	int(10)		UNSIGNED	Non	Aucune	AUTO_INCREMENT
2	name	varchar(255)	utf8_unicode_ci		Non	Aucune	
3	email	varchar(255)	utf8_unicode_ci		Non	Aucune	
4	password	varchar(60)	utf8_unicode_ci		Non	Aucune	
5	admin	tinyint(1)			Non	0	
6	remember_token	varchar(100)	utf8_unicode_ci		Oui	NULL	
7	created_at	timestamp			Non	0000-00-00 00:00:00	
8	updated_at	timestamp			Non	0000-00-00 00:00:00	

La table des utilisateurs

Remarquez que le modèle `User` est déjà présent dans le dossier `app` quand vous installez Laravel, parce qu'il est un peu particulier comme nous le verrons lorsque nous parlerons de l'authentification. Pour le moment, nous allons nous satisfaire du fait qu'il existe déjà sans nous soucier de son contenu.

Une ressource

Création

On crée maintenant une ressource avec `Artisan` :

```
php artisan make:controller UserController --resource
Controller created successfully.
```

Vous trouvez comme résultat le contrôleur `app/Http/Controllers/UserController` :

```
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
  /**
   * Display a listing of the resource.
  *
   * @return \Illuminate\Http\Response
```

Copyright © 2016 Eyrolles.

```
     */
    public function index()
    {
        //
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param \Illuminate\Http\Request $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        //
    }

    /**
     * Display the specified resource.
     *
     * @param int $id
     * @return \Illuminate\Http\Response
     */
    public function show($id)
    {
        //
    }

    /**
     * Show the form for editing the specified resource.
     *
     * @param int $id
     * @return \Illuminate\Http\Response
     */
    public function edit($id)
    {
        //
    }

    /**
     * Update the specified resource in storage.
     *
     * @param \Illuminate\Http\Request $request
     * @param int $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, $id)
```

Copyright © 2016 Eyrolles.

```
    {
        //
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param int $id
     * @return \Illuminate\Http\Response
     */
    public function destroy($id)
    {
        //
    }
}
```

Les sept méthodes créées couvrent la gestion complète des utilisateurs :

- `index` : pour afficher la liste des utilisateurs ;
- `create` : pour envoyer le formulaire de création d'un nouvel utilisateur ;
- `store` : pour créer un nouvel utilisateur ;
- `show` : pour afficher les données d'un utilisateur ;
- `edit` : pour envoyer le formulaire de modification d'un utilisateur ;
- `update` : pour modifier les données d'un utilisateur ;
- `destroy` : pour supprimer un utilisateur.

Routes

Pour créer toutes les routes, il suffit de cette unique ligne de code :

```
<?php
Route::resource('user', 'UserController');
```

Pour connaître les routes ainsi créées, on utilisera encore `Artisan` :

```
php artisan route:list
```

Les routes de la ressource

Copyright © 2016 Eyrolles.

Vous trouvez sept routes, avec chacune une URL, qui pointent sur les sept méthodes de notre contrôleur. Notez également que chacune a aussi un nom, qui peut être utilisé par exemple pour une redirection. Nous allons à présent considérer chacune de ces routes et créer la gestion des données, les vues et le code nécessaire au niveau du contrôleur.

Contrôleur

Il nous faut à présent coder le contrôleur. Pour rester dans la démarche de bonne organisation des classes, je continuerai à limiter le contrôleur à la réception des requêtes et l'envoi des réponses. Il faudra donc injecter :

- la validation ;
- la gestion des données.

Pour la validation, on rencontre deux cas :

- la création d'un utilisateur avec vérification de l'unicité du nom et de l'adresse et la correction du mot de passe ;
- la modification d'un utilisateur, avec la même vérification d'unicité, mais en excluant l'enregistrement en cours de modification. En outre, nous n'allons pas inclure le mot de passe dans cette modification.

On aura donc besoin de deux requêtes de formulaire : une pour la création et l'autre pour la modification.

Pour la gestion, une seule classe `UserRepository` suffira.

Si on considère ces injections, voici le code du contrôleur :

```
<?php
namespace App\Http\Controllers;

use App\Http\Requests\UserCreateRequest;
use App\Http\Requests\UserUpdateRequest;
use App\Repositories\UserRepository;
use Illuminate\Http\Request;

class UserController extends Controller
{
  protected $userRepository;
  protected $nbrPerPage=4;

  public function __construct(UserRepository $userRepository)
  {
    $this->userRepository=$userRepository;
  }

 public function index()
  {
    $users=$this->userRepository->getPaginate($this->nbrPerPage);
    $links=$users->render();
    return view('index', compact('users', 'links'));
  }
```

Copyright © 2016 Eyrolles.

```
  public function create()
  {
   return view('create');
  }

  public function store(UserCreateRequest $request)
  {
    $user=$this->userRepository->store($request->all());
    return redirect('user')->withOk("L'utilisateur " . $user->name . " a été
créé.");
  }

  public function show($id)
  {
    $user=$this->userRepository->getById($id);
    return view('show', compact('user'));
  }

  public function edit($id)
  {
    $user=$this->userRepository->getById($id);
    return view('edit', compact('user'));
  }

  public function update(UserUpdateRequest $request, $id)
  {
    $this->userRepository->update($id, $request->all());
    return redirect('user')->withOk("L'utilisateur " . $request-
>input('name') . " a été modifié.");
  }

  public function destroy($id)
  {
    $this->userRepository->destroy($id);
    return redirect()->back();
  }
}
```

Nous allons évidemment analyser tout cela dans le détail, mais globalement vous voyez que le code est très épuré :

- réception de la requête ;
- délégation du traitement si nécessaire (validation et gestion) ;
- envoi de la réponse.

La validation

Création d'un utilisateur

On crée une requête de formulaire pour la création d'un utilisateur. Voici le code de la classe :

Copyright © 2016 Eyrolles.

```
<?php
namespace App\Http\Requests;

use App\Http\Requests\Request;

class UserCreateRequest extends Request
{
  public function authorize()
  {
    return true;
  }

  public function rules()
  {
    return [
      'name'=>'required|max:255|unique:users',
      'email'=>'required|email|max:255|unique:users',
      'password'=>'required|confirmed|min:6'
    ];
  }
}
```

Elle contient trois règles :

- `name` : champ requis, longueur maximale de 255 caractères et unique dans la table `users` ;
- `email` : champ requis, adresse valide, longueur maximale de 255 caractères et unique dans la table `users` ;
- `password` : champ requis, longueur minimale de 6 caractères et doit correspondre à ce qui est entré dans le champ de confirmation du mot de passe.

Modification d'un utilisateur

Pour la modification d'un utilisateur, nous allons rencontrer un petit souci. En effet, on veut conserver l'unicité du nom et de l'adresse dans la base ; il est donc judicieux de prévoir une règle `unique`. Cependant, comme les valeurs du nom et de l'adresse sont déjà dans la base, on obtiendra un échec de la validation en cas de non-modification d'une de ces deux valeurs, ce qui est fortement probable. Comment nous en sortir ? Étant donné qu'on dispose d'une fonction, on peut effectuer tous les traitements qu'on veut. Voici alors la requête de formulaire pour la modification d'un utilisateur :

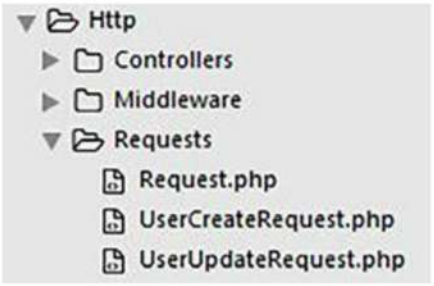
```
<?php
namespace App\Http\Requests;

use App\Http\Requests\Request;

class UserUpdateRequest extends Request
{
  public function authorize()
  {
```

Copyright © 2016 Eyrolles.

```
        return true;
    }

    public function rules()
    {
        $id=$this->user;
        return [
            'name'=>'required|max:255|unique:users,name,' . $id,
            'email'=>'required|email|max:255|unique:users,email,' . $id
        ];
    }
}
```

On récupère l'identifiant de l'utilisateur dans l'URL. Ensuite, on utilise une possibilité d'exclusion de la règle `unique`.

Vous devez donc avoir les deux fichiers suivants pour la validation.



Les fichiers de la validation

Dans le prochain chapitre, nous poursuivrons l'étude de cette ressource avec la gestion des données et les vues.

En résumé

- Lors de la migration, le constructeur de schéma permet de fixer toutes les propriétés des champs.
- Une ressource dans Laravel est constituée d'un contrôleur comportant les sept méthodes nécessaires à une gestion complète.
- Les routes vers une ressource sont créées avec une simple ligne de code.
- Pour une ressource, la validation est toujours différente entre la création et la modification. Il faut adapter le code pour en tenir compte.

Copyright © 2016 Eyrolles.

Copyright © 2016 Eyro les.

12 Ressources pour les utilisateurs et erreurs

Dans ce chapitre, nous allons poursuivre notre étude de la ressource pour les utilisateurs. Nous avons au chapitre précédent passé en revue la migration, les routes, le contrôleur et la validation. Il nous reste à aborder la gestion des données, les vues et aussi comment tout cela s'articule pour fonctionner.

Le gestionnaire de données (repository)

Nous avons vu que nous injectons un gestionnaire de données dans le contrôleur en plus des deux classes de validation :

```
<?php
public function __construct(UserRepository $userRepository)
{
  $this->userRepository=$userRepository;
}
```

Ce gestionnaire est chargé de toutes les actions au niveau de la table des utilisateurs. Voici son code (`app/Repositories/UserRepository.php`) :

```
<?php
namespace App\Repositories;

use App\User;

class UserRepository
{
  protected $user;
```

Copyright © 2016 Eyrolles.

```
  public function __construct(User $user)
  {
    $this->user=$user;
  }

  private function save(User $user, Array $inputs)
  {
    $user->name=$inputs['name'];
    $user->email=$inputs['email'];
    $user->admin=isset($inputs['admin']);
    $user->save();
  }

  public function getPaginate($n)
  {
    return $this->user->paginate($n);
  }

  public function store(Array $inputs)
  {
    $user=new $this->user;
    $user->password=bcrypt($inputs['password']);
    $this->save($user, $inputs);
    return $user;
  }

  public function getById($id)
  {
    return $this->user->findOrFail($id);
  }

  public function update($id, Array $inputs)
  {
    $this->save($this->getById($id), $inputs);
  }

  public function destroy($id)
  {
    $this->getById($id)->delete();
  }
}
```

Vous devez trouver ce fichier dans le dossier `Repository`.

Les fichiers pour la gestion

Nous allons à présent analyser ses différentes actions.

Copyright © 2016 Eyrolles.

getPaginate

Cette méthode contient une seule ligne :

```
<?php
public function getPaginate($n)
{
  return $this->user->paginate($n);
}
```

Elle est appelée depuis la méthode `index` du contrôleur :

```
<?php
public function index()
{
  $users=$this->userRepository->getPaginate($this->nbrPerPage);
  $links=$users->render();
  return view('index', compact('users', 'links'));
}
```

Cette méthode répond à l'URL *http://monsite.fr/user* (avec le verbe `get`).

On utilise la méthode `paginate` du modèle qui prend seulement une partie des enregistrements pour réaliser une pagination. Ici, j'ai prévu la valeur 4 (stockée dans la propriété `$nbrPerPage`), donc 4 enregistrements par page. Si vous regardez les requêtes SQL générées, vous trouverez :

```
select count(*) as aggregate from 'users'
select * from 'users' limit 4 offset 0
```

La première requête sert à connaître le nombre total d'enregistrements dans la table `users` (donnée nécessaire pour calculer la pagination) et la seconde à sélectionner les quatre premiers pour les afficher sur la première page. Évidemment, pour la page 2 on aura la requête suivante :

```
select * from 'users' limit 4 offset 4
```

Nous choisissons toujours 4 enregistrements, mais avec un `offset` de 4 pour prendre les 4 suivants. L'URL sera alors *.../user?page=2*.

Une fois que les enregistrements sont récupérés dans la table, on les retourne au contrôleur. Celui-ci les tranforme en un tableau avec comme clé `"users"`. Ce tableau sera utilisé par la vue comme nous le verrons bientôt.

La pagination est incluse dans la variable `$links` pour être aussi envoyée dans la vue.

Copyright © 2016 Eyrolles.

getById

Cette méthode contient juste une ligne :

```
<?php
public function getById($id)
{
  return $this->user->findOrFail($id);
}
```

La méthode `findOrFail` essaie de récupérer dans la table l'enregistrement dont on transmet l'`id`. Si elle n'y parvient pas, elle crée une erreur d'exécution.

L'erreur générée est `ModelNotFoundException`. Nous allons bientôt expliquer comment on gère les erreurs.

Voici le genre de requête lancée :

```
select * from 'users' where 'id'='2' limit 1
```

Dans le contrôleur, nous avons deux méthodes qui utilisent `getById`.

show

Voici la méthode `show` du contrôleur :

```
<?php
public function show($id)
{
  $user=$this->userRepository->getById($id);
  return view('show', compact('user'));
}
```

Cette méthode répond à l'URL *http://monsite.fr/user/n* (avec le verbe `get`), où `n` est l'identifiant de l'utilisateur qu'on veut afficher.

Une fois que l'enregistrement est ainsi récupéré dans la table, on le retourne sous la forme d'un tableau avec comme clé `"user"`. Ce tableau sera utilisé par la vue comme nous le verrons bientôt.

edit

La méthode `getById` est aussi utilisée par la méthode `edit` du contrôleur. Cette méthode répond à l'URL *http://monsite.fr/user/n/edit* (avec le verbe `get`) où `n` est l'identifiant de l'utilisateur qu'on veut modifier.

Copyright © 2016 Eyrolles.

Elle contient juste deux lignes :

```
<?php
public function edit($id)
{
  $user=$this->userRepository->getById($id);
  return view('edit', compact('user'));
}
```

Le principe est exactement le même que pour la méthode `show`.

update

Cette méthode est destinée à mettre à jour l'enregistrement dans la table à partir des données transmises comme paramètres :

```
<?php
public function update($id, Array $inputs)
{
  $this->save($this->getById($id), $inputs);
}
```

On récupère l'enregistrement par la méthode `getById` avec l'`id` transmis. Ensuite, on met à jour dans la table avec la méthode privée `save` :

```
<?php
private function save(User $user, Array $inputs)
{
  $user->name=$inputs['name'];
  $user->email=$inputs['email'];
  $user->admin=isset($inputs['admin']);
  $user->save();
}
```

La méthode `update` du gestionnaire est appelée depuis celle du contrôleur :

```
<?php
public function update(UserUpdateRequest $request, $id)
{
  $this->userRepository->update($id, $request->all());
  return redirect('user')->withOk("L'utilisateur " . $request->input('name')
. " a été modifié.");
}
```

Cette méthode répond à l'URL *http://monsite.fr/user/n* (avec le verbe `put`) où `n` est l'identifiant de l'utilisateur qu'on veut modifier.

Copyright © 2016 Eyrolles.

store

Voici le code de cette méthode :

```
<?php
public function store(Array $inputs)
{
  $user=new $this->user;
  $user->password=bcrypt($inputs['password']);
  $this->save($user, $inputs);
  return $user;
}
```

Par sécurité, le mot de passe entré est ici codé avec l'*helper* `bcrypt`. Ainsi, il ne sera pas inscrit en clair dans la table, mais sous forme codée.

On crée un nouvel objet `User`. On renseigne l'attribut `password` et on enregistre dans la table avec la méthode privée `save`.

La méthode `store` du gestionnaire est appelée depuis celle du contrôleur :

```
<?php
public function store(UserCreateRequest $request)
{
  $user=$this->userRepository->store($request->all());
  return redirect('user')->withOk("L'utilisateur " . $user->name . " a été
créé.");
}
```

Cette méthode répond à l'URL *http://monsite.fr/user* (avec le verbe `post`).

destroy

Cette méthode contient juste une ligne :

```
<?php
public function destroy($id)
{
  $this->getById($id)->delete();
}
```

Elle est appelée depuis la méthode `destroy` du contrôleur :

```
<?php
public function destroy($id)
{
  $this->userRepository->destroy($id);
  return redirect()->back();
}
```

Copyright © 2016 Eyrolles.

On supprime un enregistrement avec la méthode `delete` du modèle. Remarquez la redirection dans le contrôleur avec la méthode `back`. On renvoie ainsi la dernière requête.

Cette méthode répond à l'URL *http://monsite.fr/user/n* (avec le verbe `delete`) où `n` est l'identifiant de l'enregistrement qu'on veut supprimer.

Les vues

Étudions à présent les vues pour l'interaction avec le client. J'ai adopté pour les vues le même nom que les méthodes appelantes du contrôleur pour faciliter la compréhension.

Template

Nous aurons le même template pour toutes les vues, celui que nous avons déjà utilisé dans les précédents chapitres :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Mon joli site</title>
    {!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/
bootstrap.min.css') !!}
    {!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/
bootstrap-theme.min.css') !!}
    <!--[if lt IE 9]>
      {{ Html::style('https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.
js') }}
      {{ Html::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.
min.js') }}
    <![endif]-->
    <style>textarea {resize:none;}</style>
  </head>
  <body>
    @yield('contenu')
  </body>
</html>
```

Vue index

Cette vue est destinée à afficher la liste paginée des utilisateurs avec des boutons pour pouvoir accomplir toutes les actions. Voici le code de cette vue :

Copyright © 2016 Eyrolles.

```
@extends('template')

@section('contenu')
  <br>
  <div class="col-sm-offset-4 col-sm-4">
    @if(session()->has('ok'))
      <div class="alert alert-success alert-
dismissible">{!! session('ok') !!}</div>
    @endif
    <div class="panel panel-primary">
      <div class="panel-heading">
        <h3 class="panel-title">Liste des utilisateurs</h3>
      </div>
      <table class="table">
        <thead>
          <tr>
            <th>#</th>
            <th>Nom</th>
            <th></th>
            <th></th>
            <th></th>
          </tr>
        </thead>
        <tbody>
          @foreach ($users as $user)
            <tr>
              <td>{!! $user->id !!}</td>
              <td class="text-primary"><strong>{!! $user->name !!}</
strong></td>
              <td>{!! link_to_route('user.show', 'Voir', [$user->id],
['class'=>'btn btn-success btn-block']) !!}</td>
              <td>{!! link_to_route('user.edit', 'Modifier', [$user->id],
['class'=>'btn btn-warning btn-block']) !!}</td>
              <td>{!! Form::open(['method'=>'DELETE', 'route'=>['user.
destroy', $user->id]]) !!}
                  {!! Form::submit('Supprimer', ['class'=>'btn btn-
danger btn-block', 'onclick'=>'return confirm(\'Vraiment supprimer cet
utilisateur ?\')']) !!}
                {!! Form::close() !!}
              </td>
            </tr>
          @endforeach
        </tbody>
      </table>
    </div>
    {!! link_to_route('user.create', 'Ajouter un utilisateur', [],
['class'=>'btn btn-info pull-right']) !!}
    {!! $links !!}
  </div>
@endsection
```

Copyright © 2016 Eyrolles.

La vue index

Remarquez que pour générer la méthode `delete`, on est obligé de créer un formulaire.

La vue teste aussi la présence d'une variable `'ok'` dans la session et affiche le message correspondant dans une barre si c'est le cas.

Vue show

La vue `show` sert à afficher la fiche d'un utilisateur avec son nom, son adresse électronique et son appartenance éventuelle au groupe des administrateurs :

```
@extends('template')

@section('contenu')
  <div class="col-sm-offset-4 col-sm-4">
    <br>
    <div class="panel panel-primary">
      <div class="panel-heading">Fiche d'utilisateur</div>
      <div class="panel-body">
        <p>Nom : {{$user->name}}</p>
        <p>Email : {{$user->email}}</p>
        @if($user->admin==1)
          Administrateur
        @endif
      </div>
    </div>
    <a href="javascript:history.back()" class="btn btn-primary">
      <span class="glyphicon glyphicon-circle-arrow-left"></span>Retour
    </a>
  </div>
@endsection
```

Copyright © 2016 Eyrolles.

La vue show

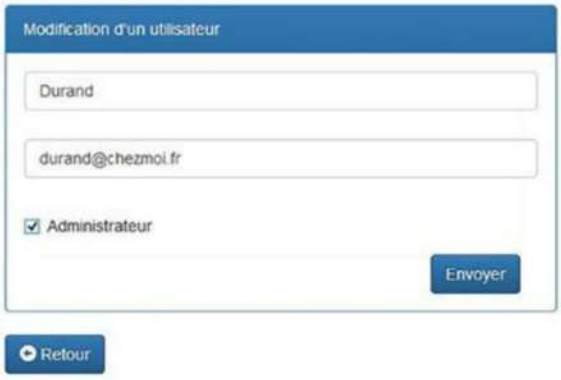
Vue edit

La vue `edit` sert à la modification d'un utilisateur. Elle affiche un formulaire :

```
@extends('template')

@section('contenu')
  <div class="col-sm-offset-4 col-sm-4">
    <br>
    <div class="panel panel-primary">
      <div class="panel-heading">Modification d'un utilisateur</div>
      <div class="panel-body">
        <div class="col-sm-12">
          {!! Form::model($user, ['route'=>['user.update', $user->id],
'method'=>'put', 'class'=>'form-horizontal panel']) !!}
        <div class="form-group {!! $errors->has('name') ? 'has-
error' : '' !!}">
            {!! Form::text('name', null, ['class'=>'form-control',
'placeholder'=>'Nom']) !!}
            {!! $errors->first('name', '<small class="help-block">:message</
small>') !!}
          </div>
          <div class="form-group {!! $errors->has('email') ? 'has-
error' : '' !!}">
            {!! Form::email('email', null, ['class'=>'form-control',
'placeholder'=>'Email']) !!}
           {!! $errors->first('email', '<small class="help-block">:message</
small>') !!}
          </div>
          <div class="form-group">
            <div class="checkbox">
              <label>
               {!! Form::checkbox('admin', 1, null) !!}Administrateur
              </label>
            </div>
          </div>
           {!! Form::submit('Envoyer', ['class'=>'btn btn-primary pull-
right']) !!}
         {!! Form::close() !!}
       </div>
```

Copyright © 2016 Eyrolles.

```
      </div>
    </div>
    <a href="javascript:history.back()" class="btn btn-primary">
      <span class="glyphicon glyphicon-circle-arrow-left"></span>Retour
    </a>
  </div>
@endsection
```



La vue edit

Remarquez l'utilisation de `Form::model` qui permet de lier le formulaire au modèle et ainsi de renseigner automatiquement les contrôles qui possèdent le même nom qu'un champ de l'enregistrement.

Vue create

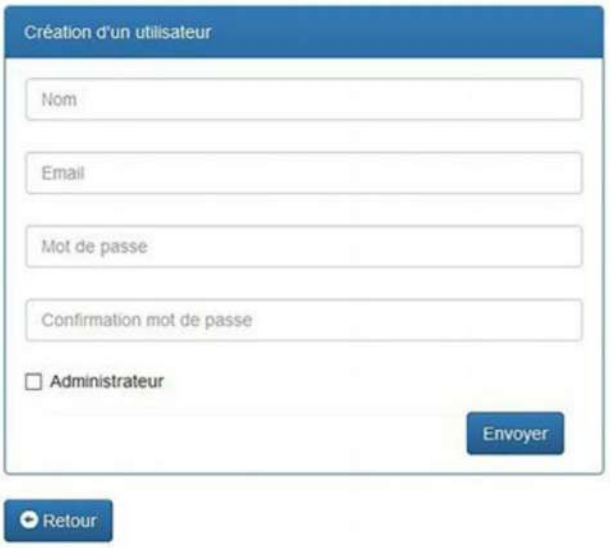
Cette vue sert à afficher le formulaire pour créer un utilisateur. C'est quasiment la même que pour la modification, avec le mot de passe en plus :

```
@extends('template')

@section('contenu')
  <div class="col-sm-offset-4 col-sm-4">
    <br>
    <div class="panel panel-primary">
      <div class="panel-heading">Création d'un utilisateur</div>
      <div class="panel-body">
        <div class="col-sm-12">
          {!! Form::open(['route'=>'user.store', 'class'=>'form-horizontal
panel']) !!}
          <div class="form-group {!! $errors->has('name') ? 'has-
error' : '' !!}">
```

Copyright © 2016 Eyrolles.

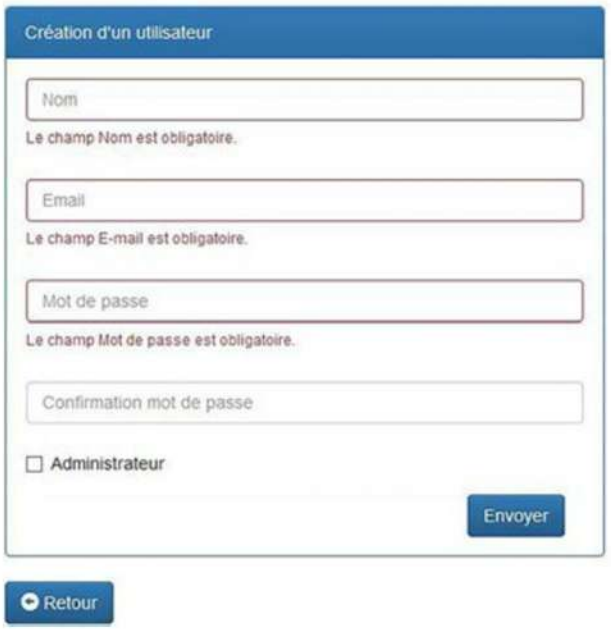
```
            {!! Form::text('name', null, ['class'=>'form-control',
'placeholder'=>'Nom']) !!}
            {!! $errors->first('name', '<small class="help-block">:message</
small>') !!}
          </div>
          <div class="form-group {!! $errors->has('email') ? 'has-
error' : '' !!}">
            {!! Form::email('email', null, ['class'=>'form-control',
'placeholder'=>'Email']) !!}
            {!! $errors->first('name', '<small class="help-block">:message</
small>') !!}
          </div>
          <div class="form-group {!! $errors->has('password') ? 'has-
error' : '' !!}">
            {!! Form::password('password', ['class'=>'form-control',
'placeholder'=>'Mot de passe']) !!}
            {!! $errors->first('password', '<small class="help-
block">:message</small>') !!}
          </div>
          <div class="form-group">
            {!! Form::password('password_confirmation', ['class'=>'form-
control', 'placeholder'=>'Confirmation mot de passe']) !!}
          </div>
          <div class="form-group">
            <div class="checkbox">
              <label>
                {!! Form::checkbox('admin', 1, null) !!} Administrateur
              </label>
            </div>
          </div>
          {!! Form::submit('Envoyer', ['class'=>'btn btn-primary pull-
right']) !!}
          {!! Form::close() !!}
        </div>
      </div>
    </div>
    <a href="javascript:history.back()" class="btn btn-primary">
      <span class="glyphicon glyphicon-circle-arrow-left"></span>Retour
    </a>
  </div>
@endsection
```

Copyright © 2016 Eyrolles.



La vue create

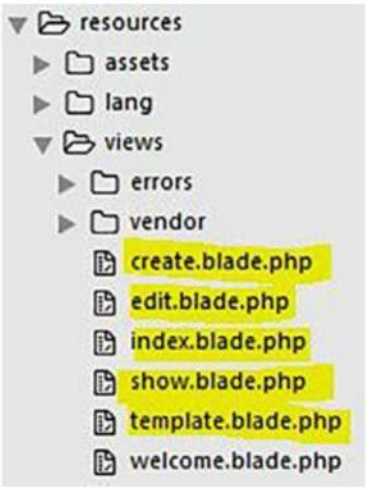
La validation est évidemment active.



La validation en action

Nous avons vu dans un chapitre précédent comment obtenir ces messages en français. Vous devez donc avoir les cinq vues suivantes.

Copyright © 2016 Eyrolles.



Les cinq vues pour la ressource

Une réflexion sur le code

Gestionnaire de base

Notre code fonctionne correctement, mais est-il vraiment performant ? Lorsqu'on crée une classe, une bonne question est de se demander si le code est réutilisable. Si on observe le gestionnaire créé pour les utilisateurs, on se rend compte qu'il est très ciblé sur le modèle concerné ; si on a une autre ressource dans l'application, il est fort probable qu'on bidouillera avec du copier-coller, ce qui est toujours source de répétition de code et d'erreurs. Est-il possible de créer un gestionnaire de base pour les ressources ? Voici une solution :

```
<?php
namespace App\Repositories;

abstract class ResourceRepository
{
  protected $model;

  public function getPaginate($n)
  {
    return $this->model->paginate($n);
  }

 public function store(Array $inputs)
  {
    return $this->model->create($inputs);
  }

 public function getById($id)
  {
```

Copyright © 2016 Eyrolles.

```
    return $this->model->findOrFail($id);
  }

  public function update($id, Array $inputs)
  {
    $this->getById($id)->update($inputs);
  }

  public function destroy($id)
  {
    $this->getById($id)->delete();
  }
}
```

Il est difficile de faire plus concis. Le seul élément qui nous indique de quel modèle il s'agit est la propriété `$model`. Le reste est tout à fait anonyme. En conséquence, le gestionnaire pour les utilisateurs est très simple à écrire :

```
<?php
namespace App\Repositories;

use App\User;

class UserRepository extends ResourceRepository
{
  public function __construct(User $user)
  {
    $this->model=$user;
  }
}
```

On rencontre toutefois quelques petits soucis. Par exemple, on a vu qu'il faut crypter le mot de passe et on réalisait cela dans le gestionnaire. On pourrait surcharger la méthode `store` pour le prévoir, mais cela casserait l'harmonie du code.

Modèle

Une autre solution plus élégante est de prévoir un *mutator* au niveau du modèle. Voyons déjà ce que nous avons dans ce modèle `User` :

```
<?php
namespace App;

use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
 /**
  * The attributes that are mass assignable.
```

Copyright © 2016 Eyrolles.

```
     *
     * @var array
     */
    protected $fillable=[
      'name', 'email', 'password',
    ];

    /**
     * The attributes excluded from the model's JSON form.
     *
     * @var array
     */
    protected $hidden=[
      'password', 'remember_token',
    ];
}
```

Je reviendrai plus en détail sur les `traits` utilisés quand on étudiera l'authentification. Pour le moment, on s'intéressera à la propriété `$fillable`. Lorsqu'on crée un enregistrement avec la méthode `create` comme on l'a prévu, il y a un risque au niveau de la sécurité. Dans le tableau transmis en paramètre, on a normalement seulement les champs qu'on désire renseigner. Cependant, si un petit malin envoie d'autres informations, elles risquent fort de se propager jusqu'à la table. Pour éviter cela, on définit dans le modèle les champs qui peuvent être mis à jour avec cette méthode (on parle de mise à jour de masse) dans la propriété `$fillable`. Comme on a aussi le champ `admin` à renseigner, il faut compléter le tableau :

```
<?php
protected $fillable=['name', 'email', 'password', 'admin'];
```

On veut également crypter le mot de passe. Il suffit de mettre en place le *mutator* suivant :

```
<?php
public function setPasswordAttribute($password)
{
  $this->attributes['password']=bcrypt($password);
}
```

Ainsi, chaque fois qu'on affectera l'attribut `password`, il passera par cette méthode et la valeur sera cryptée.

Contrôleur

Il ne nous reste plus que la gestion de la case à cocher pour l'administration. On ne peut pas la résoudre comme on l'a fait pour le mot de passe puisqu'on n'a pas toujours l'information (la case à cocher n'est transmise que si elle est cochée). Il faut donc prévoir ce traitement dans le contrôleur :

Copyright © 2016 Eyrolles.

```
<?php
namespace App\Http\Controllers;

use App\Http\Requests\UserCreateRequest;
use App\Http\Requests\UserUpdateRequest;
use App\Repositories\UserRepository;
use Illuminate\Http\Request;

class UserController extends Controller
{
  protected $userRepository;
  protected $nbrPerPage=4;

  public function __construct(UserRepository $userRepository)
  {
    $this->userRepository=$userRepository;
  }

  public function index()
  {
    $users=$this->userRepository->getPaginate($this->nbrPerPage);
    $links=$users->setPath('')->render());
    return view('index', compact('users', 'links'));
  }

  public function create()
  {
    return view('create');
  }

 public function store(UserCreateRequest $request)
  {
    $this->setAdmin($request);
    $user=$this->userRepository->store($request->all());
    return redirect('user')->withOk("L'utilisateur " . $user->name . " a été
créé.");
  }

 public function show($id)
  {
    $user=$this->userRepository->getById($id);
    return view('show', compact('user'));
  }

  public function edit($id)
  {
   $user=$this->userRepository->getById($id);
```

Copyright © 2016 Eyrolles.

```
    return view('edit', compact('user'));
  }

  public function update(UserUpdateRequest $request, $id)
  {
    $this->setAdmin($request);
    $this->userRepository->update($id, $request->all());
    return redirect('user')->withOk("L'utilisateur " . $request-
>input('name') . " a été modifié.");
  }

  public function destroy($id)
  {
    $this->userRepository->destroy($id);
    return redirect()->back();
  }

  private function setAdmin($request)
  {
    if (!$request->has('admin'))
    {
      $request->merge(['admin'=>0]);
    }
  }
}
```

C'est la fonction privée `setAdmin` qui est chargée de gérer la case à cocher. On teste qu'on a la clé `admin` dans le tableau de données et, si c'est le cas, on remplace sa valeur en utilisant la méthode `merge`.

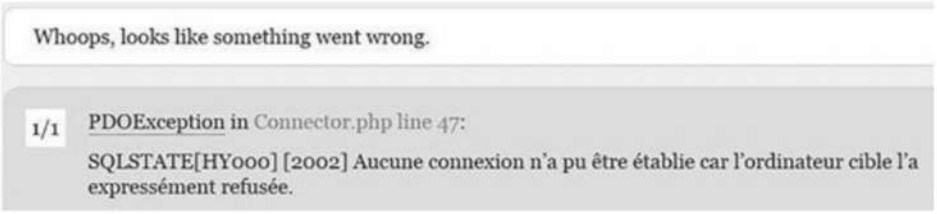
On en arrive à un code plus clair, plus facile à réutiliser et à maintenir. On pourrait pousser la réflexion au niveau du contrôleur et créer un contrôleur de base pour les ressources. Il suffirait de prévoir un préfixe pour les vues et de réfléchir aux injections ; on obtiendrait quelque chose de très élégant, mais je ne vais pas poursuivre cette réflexion pour ne pas alourdir ce chapitre. L'important est de comprendre le principe. En outre, on arrive rapidement à des impasses dans une application réelle qui oblige souvent à multiplier les codes spécifiques.

Les erreurs

La gestion des erreurs constitue une part importante dans le développement d'une application. On dit parfois que c'est dans ce domaine qu'on fait la différence entre le professionnel et l'amateur.

Puisque nous sommes dans l'accès aux données, il peut arriver un souci avec la connexion à la base. Voyons un peu ce que nous obtenons si MySQL ne répond plus… Avec notre application, si j'arrête le service de MySQL et si je lance l'URL *http://monsite.fr/user*, j'obtiens l'erreur suivante.

Copyright © 2016 Eyrolles.



Erreur suite à arrêt de MySQL

Je n'ai affiché ici que la partie supérieure. Ne soyez pas impressionné par la quantité de messages de la page d'erreurs ; avec un peu d'habitude, vous serez heureux de disposer de tous ces renseignements lorsqu'une erreur se produira dans votre application.

Cet affichage des erreurs est parfait pour la phase de développement, mais sur une application en ligne il vaut mieux cacher tout cela pour deux raisons évidentes : d'une part, cela n'intéresse pas l'utilisateur et, d'autre part, cela pourrait servir à quelqu'un de mal intentionné et lui fournir de précieux renseignements sur le fonctionnement de vos scripts.

Si vous regardez dans le fichier `.env` dont je vous ai déjà parlé, vous trouvez la ligne suivante :

```
APP_DEBUG=true
```

Si on met `false` ici pour voir la différence, on n'obtient plus que le laconique message suivant :

Le message d'erreur simplifié de Laravel

C'est trop laconique pour l'utilisateur et, surtout, cela ne lui donne aucun lien pour accéder à une autre page. Vous pouvez aussi considérer qu'un message en anglais n'est pas adapté pour votre site francophone. Malheureusement, la page correspondante se trouve dans le framework, qu'on ne va évidemment pas aller bricoler !

Le fichier des erreurs

Copyright © 2016 Eyrolles.

En voici le code :

```
<?php
namespace App\Exceptions;

use Exception;
use Illuminate\Auth\Access\AuthorizationException;
use Illuminate\Database\Eloquent\ModelNotFoundException;
use Symfony\Component\HttpKernel\Exception\HttpException;
use Illuminate\Foundation\Validation\ValidationException;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;

class Handler extends ExceptionHandler
{
  /**
   * A list of the exception types that should not be reported.
   *
   * @var array
   */
  protected $dontReport=[
    AuthorizationException::class,
    HttpException::class,
    ModelNotFoundException::class,
    ValidationException::class,
  ];

 /**
   * Report or log an exception.
   *
   * This is a great spot to send exceptions to Sentry, Bugsnag, etc.
   *
   * @param \Exception $e
   * @return void
   */
  public function report(Exception $e)
  {
    return parent::report($e);
  }

  /**
   * Render an exception into an HTTP response.
   *
   * @param \Illuminate\Http\Request $request
   * @param \Exception $e
   * @return \Illuminate\Http\Response
   */

  public function render($request, Exception $e)
  {
    return parent::render($request, $e);
  }
}
```

Toutes les erreurs d'exécution sont traitées ici. On découvre que tout est archivé avec la méthode `report`, mais où se trouve cet archivage ? Regardez dans `storage/logs` ;

Copyright © 2016 Eyrolles.

vous trouvez un fichier `laravel.log`. En l'ouvrant, vous trouvez les erreurs archivées avec leur trace. Par exemple, dans notre cas on a bien :

```
[2015-12-28 17:09:42] local.ERROR: exception 'PDOException' with message
'SQLSTATE[HY000] [2002] Aucune connexion n'a pu être établie car
l'ordinateur cible l'a expressément refusée.
```

C'est important de disposer de ce genre d'archivage des erreurs sur une application en production.

Un cas fréquent est celui de la page non trouvée (erreur 404).

L'erreur 404

Si on veut changer le message de Laravel pour le rendre à notre goût et surtout adapté à notre langue, il faut prévoir une vue, par exemple `resources/views/errors/404.blade.php` :

```
@extends('template')

@section('contenu')
  <br>
  <div class="col-sm-offset-4 col-sm-4">
    <div class="panel panel-danger">
      <div class="panel-heading">
        <h3 class="panel-title">Il y a un problème !</h3>
      </div>
      <div class="panel-body">
        <p>Nous sommes désolés, mais la page que vous désirez n'existe
pas...</p>
      </div>
    </div>
  </div>
@endsection
```

Pour une URL qui n'aboutit pas, on obtient le message suivant.

Notre vue d'erreur 404 personnalisée

Copyright © 2016 Eyrolles.

Avouez que c'est quand même mieux !

On a vu précédemment que lorsqu'on arrête le service MySQL, on génère une erreur `PDOException`. Comment intercepter cette erreur ? Voilà une solution :

```
<?php
namespace App\Exceptions;

use Exception;
use Illuminate\Auth\Access\AuthorizationException;
use Illuminate\Database\Eloquent\ModelNotFoundException;
use Symfony\Component\HttpKernel\Exception\HttpException;
use Illuminate\Foundation\Validation\ValidationException;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;

class Handler extends ExceptionHandler
{
  ...
  /**
   * Render an exception into an HTTP response.
   *
   * @param \Illuminate\Http\Request $request
   * @param \Exception $e
   * @return \Illuminate\Http\Response
   */
  public function render($request, Exception $e)
  {
    if ($e instanceof \PDOException)
    {
      return response()->view('errors.pdo', [], 500);
    }
    return parent::render($request, $e);
  }
}
```

Ainsi, on affichera une page spécifique pour cette erreur.

En résumé

- Créer un gestionnaire (*repository*) indépendant du contrôleur pour les accès aux données permet de disposer d'un code clair et facile à maintenir et tester.
- Il est important de penser à la réutilisation du code qu'on crée.
- Les vues doivent utiliser au maximum les possibilités de `Blade`, des *helpers* et de la classe `Form` pour être concises et lisibles.
- La gestion des erreurs ne doit pas être négligée. Il faut enlever le mode de débogage sur un site en production et prévoir des messages explicites pour les utilisateurs en fonction de l'erreur rencontrée.

Copyright © 2016 Eyrolles.

13 L'authentification

L'authentification constitue une tâche fréquente. Souvent en effet, certaines parties d'un site ne doivent être accessibles qu'à certains utilisateurs, ne serait-ce que l'administration. La solution proposée par Laravel est d'une grande simplicité, parce que tout est déjà préparé comme nous allons le voir dans ce chapitre.

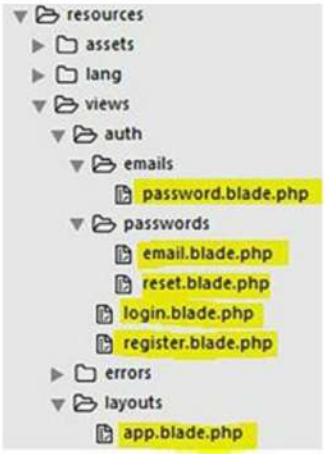
La commande Artisan

La version 5.0 de Laravel était équipée des vues, *assets* et routes pour l'authentification. L'arrivée de la version 5.1 a vu disparaître ces éléments suite à de nombreuses discussions animées. Finalement, avec la version 5.2, on n'a toujours pas tout cela de base, mais il existe une commande `Artisan` pour le construire :

```
php artisan make:auth
```

Si tout se passe bien, vous devriez obtenir les vues suivantes.

Copyright © 2016 Eyrolles.



Les vues de l'authentification

Vous devez aussi trouver un nouveau contrôleur.

Un nouveau contrôleur

Et dans le fichier des routes, un nouveau code a été ajouté :

```
<?php
Route::auth();
Route::get('/home', 'HomeController@index');
});
```

Vous êtes maintenant prêt pour ce chapitre !

`Route::auth()` crée automatiquement toutes les routes de l'authentification.

Les tables

users

Nous allons utiliser la table `users` du chapitre précédent. Par défaut, Laravel considère que cette table existe et il s'en sert comme référence pour l'authentification.

Par défaut également, c'est `Eloquent` qui est utilisé ; il est aussi possible de changer ce fonctionnement.

Copyright © 2016 Eyrolles.

password_reset

Lors de l'installation, deux migrations sont automatiquement ajoutées.

Les deux migrations présentes à l'installation

Lors des précédents chapitres, je vous ai fait supprimer la seconde parce qu'on avait juste besoin de la table des utilisateurs, mais ici elle nous sert pour la réinitialisation des mots de passe.

Lançons une commande d'`Artisan` pour supprimer toutes les tables précédemment créées :

```
php artisan migrate:reset
```

Il ne devrait alors plus vous rester que la table `migrations` vide. Si ce n'est pas le cas, faites le nécessaire dans votre base.

Avec les deux migrations présentes, lancez alors la commande d'`Artisan` pour créer les tables :

```
php artisan migrate
```

Vous devriez normalement obtenir trois tables.

Les trois tables

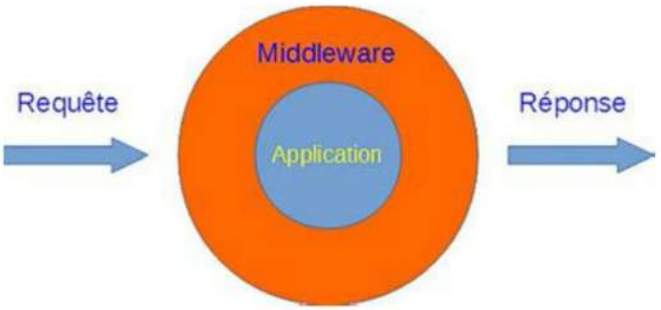
Les tables sont prêtes, passons donc à la suite.

Conservez la migration des précédents chapitres pour les utilisateurs avec la colonne `admin`. N'utilisez pas celle fournie de base avec Laravel, qui ne comporte pas cette colonne.

Copyright © 2016 Eyrolles.

Les middlewares

Qu'est-ce qu'un middleware ?



Positionnement fonctionnel des middlewares

Un *middleware* effectue un traitement à l'arrivée de la requête ou à son départ. Par exemple, la gestion des sessions ou des cookies dans Laravel se fait dans un *middleware*, ainsi que l'authentification. On a en fait plusieurs *middlewares* en couches successives, chacun effectuant son traitement et transmetant la requête ou la réponse au suivant.

Laravel s'installe avec deux fichiers *middlewares* qui concernent l'authentification.

Les middlewares pour l'authentification

Authenticate

Ce *middleware* indique si un utilisateur est authentifié. Voici son code :

```
<?php
namespace App\Http\Middleware;

use Closure;
use Illuminate\Support\Facades\Auth;

class Authenticate
{
```

Copyright © 2016 Eyrolles.

```
    /**
     * Handle an incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @param string|null $guard
     * @return mixed
     */
    public function handle($request, Closure $next, $guard=null)
    {
        if (Auth::guard($guard)->guest()) {
            if ($request->ajax() || $request->wantsJson()) {
                return response('Unauthorized.', 401);
            } else {
                return redirect()->guest('login');
            }
        }
        return $next($request);
    }
}
```

On regarde si l'utilisateur est juste un invité (`guest`). Si c'est le cas, on teste si la requête est en Ajax, auquel cas on renvoie `Unauthorized`. Si elle n'est pas en Ajax, on redirige vers la route `login` pour que l'invité puisse s'authentifier. Si ce n'est pas un invité, on laisse la requête suivre normalement son cours. Tout cela n'est évidemment pas figé dans le marbre et on peut changer tout ce qu'on veut. Pour cet ouvrage, je me contenterai de prendre le code par défaut.

La méthode `redirect()->guest` équivaut à la méthode `redirect()` avec la différence que l'URL est mémorisée en session pour pouvoir rediriger l'utilisateur à l'issue de son authentification vers l'URL qu'il voulait initialement atteindre.

RedirectIfAuthenticated

Ce *middleware* indique si l'utilisateur n'est pas authentifié. C'est donc l'exact inverse du précédent. Voici son code :

```
<?php
namespace App\Http\Middleware;

use Closure;
use Illuminate\Support\Facades\Auth;

class RedirectIfAuthenticated
{
```

Copyright © 2016 Eyrolles.

```
    /**
     * Handle an incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @param string|null $guard
     * @return mixed
     */
    public function handle($request, Closure $next, $guard=null)
    {
        if (Auth::guard($guard)->check()) {
            return redirect('/');
        }
        return $next($request);
    }
}
```

Si l'utilisateur est authentifié, ce qui nous est indiqué par la méthode `check`, alors on renvoie à la page de base du site (`/`). Sinon, on laisse la requête suivre normalement son cours.

Routes et contrôleurs

Routes

On a vu qu'on a créé des routes avec `Artisan` :

```
<?php
Route::auth();
Route::get('/home', 'HomeController@index');
```

Comme ce n'est pas très explicite, voyons un peu les routes ainsi créées.

```
GET|HEAD | home
GET|HEAD | login
POST     | login
GET|HEAD | logout
POST     | password/email
POST     | password/reset
GET|HEAD | password/reset/{token?}
GET|HEAD | register
POST     | register
```

```
App\Http\Controllers\HomeController@index                          | web,auth
App\Http\Controllers\Auth\AuthController@showLoginForm             | web,guest
App\Http\Controllers\Auth\AuthController@login                     | web,guest
App\Http\Controllers\Auth\AuthController@logout                    | web
App\Http\Controllers\Auth\PasswordController@sendResetLinkEmail    | web,guest
App\Http\Controllers\Auth\PasswordController@reset                 | web,guest
App\Http\Controllers\Auth\PasswordController@showResetForm         | web,guest
App\Http\Controllers\Auth\AuthController@showRegistrationForm      | web,guest
App\Http\Controllers\Auth\AuthController@register                  | web,guest
```

Les routes créées

Copyright © 2016 Eyrolles.

Contrôleurs

Il est fait référence à deux contrôleurs.

Les deux contrôleurs de l'authentification

AuthController

Ce contrôleur est destiné à gérer :

- l'enregistrement des utilisateurs ;
- la connexion ;
- la déconnexion.

Voici son code :

```
<?php
namespace App\Http\Controllers\Auth;

use App\User;
use Validator;
use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\ThrottlesLogins;
use Illuminate\Foundation\Auth\AuthenticatesAndRegistersUsers;

class AuthController extends Controller
{
  /*
  |--------------------------------------------------------------------------
  | Registration & Login Controller
  |--------------------------------------------------------------------------
  |
  | This controller handles the registration of new users, as well as the
  | authentication of existing users. By default, this controller uses
  | a simple trait to add these behaviors. Why don't you explore it ?
  |
  */

  use AuthenticatesAndRegistersUsers, ThrottlesLogins;
  /**
   * Where to redirect users after login/registration.
   *
   * @var string
   */
  protected $redirectTo='/';
```

Copyright © 2016 Eyrolles.

```
    /**
     * Create a new authentication controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware($this->guestMiddleware(), ['except'=>'logout']);
    }
    /**
     * Get a validator for an incoming registration request.
     *
     * @param array $data
     * @return \Illuminate\Contracts\Validation\Validator
     */
    protected function validator(array $data)
    {
        return Validator::make($data, [
            'name'=>'required|max:255',
            'email'=>'required|email|max:255|unique:users',
            'password'=>'required|min:6|confirmed',
        ]);
    }

    /**
     * Create a new user instance after a valid registration.
     *
     * @param array $data
     * @return User
     */
    protected function create(array $data)
    {
        return User::create([
            'name'=>$data['name'],
            'email'=>$data['email'],
            'password'=>bcrypt($data['password']),
        ]);
    }
}
```

On ne retrouve aucune des méthodes dont il est fait référence dans les routes. Vous pouvez remarquer l'utilisation du `trait Illuminate\Foundation\Auth\AuthenticatesAndRegistersUsers`. Donc, toutes les méthodes se trouvent dans le framework lui-même. Si nous avons besoin de les personnaliser, la seule solution est donc de les surcharger.

Pour mémoire, un `trait` est destiné à ajouter des fonctionnalités à une classe sans passer par l'héritage.

Le `trait ThrottlesLogins` est destiné à mettre en place une protection contre les attaques « brute force ».

Copyright © 2016 Eyrolles.

PasswordController

Ce contrôleur est destiné uniquement à permettre la réinitialisation du mot de passe en cas d'oubli par l'utilisateur. Voici son code :

```
<?php
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\ResetsPasswords;

class PasswordController extends Controller
{
  /*
  |--------------------------------------------------------------------------
  | Password Reset Controller
  |--------------------------------------------------------------------------
  |
  | This controller is responsible for handling password reset requests
  | and uses a simple trait to include this behavior. You're free to
  | explore this trait and override any methods you wish to tweak.
  |
  */

  use ResetsPasswords;

  /**
   * Create a new password controller instance.
   *
   * @return void
   */
  public function __construct()
  {
    $this->middleware('guest');
  }
}
```

On se rend compte que lui aussi utilise un `trait` (`Illuminate\Foundation\Auth\ResetsPasswords`).

Les vues

Les vues de l'authentification

Copyright © 2016 Eyrolles.

Vous n'êtes évidemment pas obligés de garder ces vues si vous voulez assurer la cohérence visuelle de votre site, mais pour ce chapitre on les utilisera directement. Elles sont toutes conçues de la même manière. Voici par exemple la vue pour la connexion (`resources/views/auth/login.blade.php`) :

```
@extends('layouts.app')

@section('content')
<div class="container">
  <div class="row">
    <div class="col-md-8 col-md-offset-2">
      <div class="panel panel-default">
        <div class="panel-heading">Login</div>
        <div class="panel-body">
          <form class="form-horizontal" role="form" method="POST" action="{{
url('/login') }}">
            {!! csrf_field() !!}
            <div class="form-group{{ $errors->has('email') ? ' has-
error' : '' }}">
              <label class="col-md-4 control-label">E-Mail Address</label>
              <div class="col-md-6">
                <input type="email" class="form-control" name="email"
value="{{ old('email') }}">
                @if ($errors->has('email'))
                  <span class="help-block">
                    <strong>{{ $errors->first('email') }}</strong>
                  </span>
                @endif
              </div>
            </div>

            <div class="form-group{{ $errors->has('password') ? ' has-
error' : '' }}">
              <label class="col-md-4 control-label">Password</label>
              <div class="col-md-6">
                <input type="password" class="form-control" name="password">
                @if ($errors->has('password'))
                  <span class="help-block">
                    <strong>{{ $errors->first('password') }}</strong>
                  </span>
                @endif
              </div>
            </div>
          <div class="form-group">
            <div class="col-md-6 col-md-offset-4">
              <div class="checkbox">
                <label>
                  <input type="checkbox" name="remember">Remember Me
                </label>
              </div>
            </div>
          </div>

          <div class="form-group">
            <div class="col-md-6 col-md-offset-4">
              <button type="submit" class="btn btn-primary">
```

Copyright © 2016 Eyrolles.

```
                    <i class="fa fa-btn fa-sign-in"></i>Login
                  </button>

                  <a class="btn btn-link" href="{{ url('/password/
reset') }}">Forgot Your Password?</a>
                </div>
              </div>
            </form>
          </div>
        </div>
      </div>
    </div>
</div>
@endsection
```

Évidemment, tout est en anglais ! De plus, on voit qu'on utilise un template (`app`).

L'enregistrement d'un utilisateur

Validation

Dans le contrôleur `AuthController`, vous trouvez le code suivant :

```
<?php
protected function validator(array $data)
{
  return Validator::make($data, [
    'name'=>'required|max:255',
    'email'=>'required|email|max:255|unique:users',
    'password'=>'required|min:6|confirmed',
  ]);
}
```

On a vu jusqu'à présent la validation se faire à partir d'une requête de formulaire ; là, elle est réalisée différemment. Il était sans doute difficile pour le framework de gérer correctement les espaces de noms dans ce cas. On trouve aussi dans cette classe une méthode pour créer l'utilisateur :

```
<?php
protected function create(array $data)
{
  return User::create([
    'name'=>$data['name'],
    'email'=>$data['email'],
    'password'=>bcrypt($data['password']),
  ]);
}
```

Copyright © 2016 Eyrolles.

Comme on a ajouté une colonne `admin`, il faut un peu modifier le code pour l'enregistrer aussi :

```
<?php
public function create(array $data)
{
  return User::create([
    'name'=>$data['name'],
    'email'=>$data['email'],
    'password'=>bcrypt($data['password']),
    'admin'=>isset($data['admin'])
  ]);
}
```

Attention, dans le chapitre précédent on a mis en place dans le modèle `User` un *mutator* pour crypter automatiquement le mot de passe. Si vous le cryptez deux fois, vous allez avoir des soucis ! Il faut donc choisir la méthode que vous voulez utiliser.

Contrôleur

Avant d'envisager une authentification, un visiteur doit pouvoir s'enregistrer.

Si on regarde le contenu du `trait AuthenticatesAndRegistersUsers`, on se rend compte qu'il fait appel à deux autres `traits` :

```
<?php
namespace Illuminate\Foundation\Auth;

trait AuthenticatesAndRegistersUsers
{
  use AuthenticatesUsers, RegistersUsers {
    AuthenticatesUsers::redirectPath insteadof RegistersUsers;
    AuthenticatesUsers::getGuard insteadof RegistersUsers;
  }
}
```

Il y a deux méthodes dans le `trait RegistersUsers` pour l'enregistrement des utilisateurs :

```
<?php
namespace Illuminate\Foundation\Auth;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

trait RegistersUsers
{
  use RedirectsUsers;
```

Copyright © 2016 Eyrolles.

```
    /**
     * Show the application registration form.
     *
     * @return \Illuminate\Http\Response
     */
    public function getRegister()
    {
        return $this->showRegistrationForm();
    }

    /**
     * Show the application registration form.
     *
     * @return \Illuminate\Http\Response
     */
    public function showRegistrationForm()
    {
        if (property_exists($this, 'registerView')) {
            return view($this->registerView);
        }
        return view('auth.register');
    }

    /**
     * Handle a registration request for the application.
     *
     * @param \Illuminate\Http\Request $request
     * @return \Illuminate\Http\Response
     */
    public function postRegister(Request $request)
    {
        return $this->register($request);
    }

    /**
     * Handle a registration request for the application.
     *
     * @param \Illuminate\Http\Request $request
     * @return \Illuminate\Http\Response
     */
    public function register(Request $request)
    {
        $validator=$this->validator($request->all());
        if ($validator->fails()) {
            $this->throwValidationException(
                $request, $validator
            );
        }
        Auth::guard($this->getGuard())->login($this->create($request->all()));
        return redirect($this->redirectPath());
    }
```

Copyright © 2016 Eyrolles.

```
    /**
     * Get the guard to be used during registration.
     *
     * @return string|null
     */
    protected function getGuard()
    {
        return property_exists($this, 'guard') ? $this->guard:null;
    }
}
```

Voyons de plus près ces deux méthodes.

- `getRegister` : ici, on renvoie la vue `auth.register` qui doit contenir le formulaire pour l'enregistrement (notez qu'on teste la présence éventuelle d'une propriété `registerView` qui permet de changer facilement la localisation et le nom de la vue en créant cette propriété).
- `postRegister` : ici, on traite la soumission du formulaire. La validation est assurée par la méthode `validator`. Si la validation est correcte, on crée dans la base cet utilisateur avec la méthode `create`, on connecte le nouvel utilisateur avec la méthode `login`, enfin on renvoie à l'URL définie dans la méthode `redirectPath`.

Comme tout ce code se situe dans le framework, nous ne devons pas directement le modifier. On peut toutefois modifier l'URL de redirection. Regardez la méthode `redirectPath` placée dans le `trait RedirectUsers` :

```
<?php
public function redirectPath()
{
    if (property_exists($this, 'redirectPath'))
    {
        return $this->redirectPath;
    }
    return property_exists($this, 'redirectTo') ? $this->redirectTo:'/home';
}
```

On teste la présence éventuelle d'une propriété `redirectPath`. Si elle se confirme, on l'utilise pour la redirection. On teste aussi la présence d'une propriété `redirectTo`, sinon on redirige vers `home`. Donc, si on veut une redirection spécifique, il suffit de créer une propriété `redirectPath` ou `redirectTo` dans le contrôleur. Pour toute autre modification, on devra surcharger les méthodes.

L'URL est *.../register*.

Vue auth.register

Il nous faut une vue pour l'enregistrement. On a vu que Laravel nous en propose une. Voici l'aspect normalement obtenu.

Copyright © 2016 Eyrolles.

Le formulaire pour l'enregistrement

Lorsqu'un utilisateur est créé et connecté, il est renvoyé sur la route définie par la fonction `redirectPath`. Comme la route, le contrôleur et la vue sont prévus, on obtient ce qui suit.

Utilisateur enregistré et connecté

Connexion et déconnexion

Vu que les utilisateurs peuvent s'enregistrer, passons à la connexion. Par défaut, Laravel prévoit de le faire à partir de l'adresse électronique. On va conserver ce comportement.

Connexion

On a deux méthodes concernées dans le `trait AuthenticatesUsers` :

```
<?php

namespace Illuminate\Foundation\Auth;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\Lang;
```

Copyright © 2016 Eyrolles.

```
trait AuthenticatesUsers
{
    use RedirectsUsers;

    /**
     * Show the application login form.
     *
     * @return \Illuminate\Http\Response
     */
    public function getLogin()
    {
        return $this->showLoginForm();
    }

    /**
     * Show the application login form.
     *
     * @return \Illuminate\Http\Response
     */
    public function showLoginForm()
    {
        $view = property_exists($this, 'loginView')
                    ? $this->loginView : 'auth.authenticate';

        if (view()->exists($view)) {
            return view($view);
        }
        return view('auth.login');
    }

    /**
     * Handle a login request to the application.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function postLogin(Request $request)
    {
        return $this->login($request);
    }

   /**
     * Handle a login request to the application.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function login(Request $request)
    {
        $this->validateLogin($request);

       // If the class is using the ThrottlesLogins trait, we can
automatically throttle
       // the login attempts for this application. We'll key this by the
username and
       // the IP address of the client making these requests into this
application.
```

Copyright © 2016 Eyrolles.

```
        $throttles = $this->isUsingThrottlesLoginsTrait();

        if ($throttles && $lockedOut = $this-
>hasTooManyLoginAttempts($request)) {
            $this->fireLockoutEvent($request);

            return $this->sendLockoutResponse($request);
        }

        $credentials = $this->getCredentials($request);

        if (Auth::guard($this->getGuard())->attempt($credentials, $request-
>has('remember'))) {
            return $this->handleUserWasAuthenticated($request, $throttles);
        }

        // If the login attempt was unsuccessful we will increment the
number of attempts
        // to login and redirect the user back to the login form. Of course,
when this
        // user surpasses their maximum number of attempts they will get
locked out.
        if ($throttles && ! $lockedOut) {
            $this->incrementLoginAttempts($request);
        }

        return $this->sendFailedLoginResponse($request);
    }

    /**
     * Validate the user login request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return void
     */
    protected function validateLogin(Request $request)
    {
        $this->validate($request, [
            $this->loginUsername() => 'required', 'password' => 'required',
        ]);
    }

    /**
     * Send the response after the user was authenticated.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  bool  $throttles
     * @return \Illuminate\Http\Response
     */
    protected function handleUserWasAuthenticated(Request $request,
$throttles)
    {
        if ($throttles) {
            $this->clearLoginAttempts($request);
        }
```

Copyright © 2016 Eyrolles.

```
        if (method_exists($this, 'authenticated')) {
            return $this->authenticated($request, Auth::guard($this-
>getGuard())->user());
        }

        return redirect()->intended($this->redirectPath());
    }

    /**
     * Get the failed login response instance.
     *
     * @param \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    protected function sendFailedLoginResponse(Request $request)
    {
        return redirect()->back()
            ->withInput($request->only($this->loginUsername(), 'remember'))
            ->withErrors([
                $this->loginUsername() => $this->getFailedLoginMessage(),
            ]);
    }
    /**
     * Get the failed login message.
     *
     * @return string
     */
    protected function getFailedLoginMessage()
    {
        return Lang::has('auth.failed')
                ? Lang::get('auth.failed')
                : 'These credentials do not match our records.';
    }

    /**
     * Get the needed authorization credentials from the request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    protected function getCredentials(Request $request)
    {
        return $request->only($this->loginUsername(), 'password');
    }

    /**
     * Log the user out of the application.
     *
     * @return \Illuminate\Http\Response
     */
    public function getLogout()
    {
        return $this->logout();
    }
```

Copyright © 2016 Eyrolles.

```
    /**
     * Log the user out of the application.
     *
     * @return \Illuminate\Http\Response
     */
    public function logout()
    {
        Auth::guard($this->getGuard())->logout();

        return redirect(property_exists($this, 'redirectAfterLogout') ?
$this->redirectAfterLogout : '/');
    }

    /**
     * Get the guest middleware for the application.
     */
    public function guestMiddleware()
    {
        $guard = $this->getGuard();

        return $guard ? 'guest:'.$guard : 'guest';
    }

    /**
     * Get the login username to be used by the controller.
     *
     * @return string
     */
    public function loginUsername()
    {
        return property_exists($this, 'username') ? $this->username :
'email';
    }

    /**
     * Determine if the class is using the ThrottlesLogins trait.
     *
     * @return bool
     */
    protected function isUsingThrottlesLoginsTrait()
    {
        return in_array(
            ThrottlesLogins::class, class_uses_recursive(static::class)
        );
    }

    /**
     * Get the guard to be used during authentication.
     *
     * @return string|null
     */
    protected function getGuard()
    {
        return property_exists($this, 'guard') ? $this->guard : null;
    }
}
```

Copyright © 2016 Eyrolles.

Voyons de plus près ces deux méthodes.

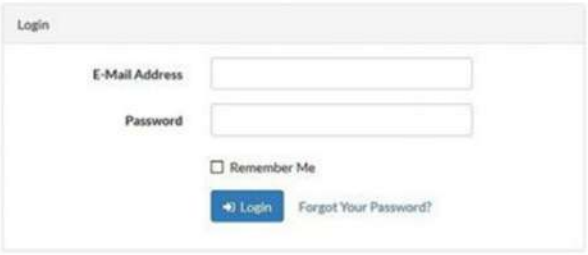
- `getLogin` : on renvoie la vue `auth.login` (sauf s'il existe une vue `auth.authenticate` ou si on a créé une propriété `loginView`) qui doit contenir le formulaire pour la connexion.
- `postLogin` : on traite la soumission du formulaire. La validation est assurée directement dans la méthode avec la puissante méthode `validate` qui est une alternative intéressante aux requêtes de formulaires. Si la validation est correcte, on s'assure qu'on n'a pas d'attaque (`throttle`), puis les données sont vérifiées dans la table avec la méthode `attempt`. Si tout va bien, on renvoie comme défini par la méthode `handleUserWasAuthenticated`, sinon on redirige vers le formulaire avec un message d'erreur défini par la fonction `getFailedLoginMessage`.

Au fil des évolutions, de nombreuses fonctions ont été mises en place pour éviter de surcharger le contrôleur ; pour chaque cas, il convient de bien regarder le code pour déterminer la meilleure stratégie à adopter.

L'URL est *.../login*.

Vue auth.login

Cette vue est aussi prévue. Voici l'aspect du formulaire de connexion :



Le formulaire de connexion

S'il y a déjà un utilisateur connecté, vous allez être piégé par le *middleware* pour accéder à ce formulaire. Déconnectez l'utilisateur par le menu ou supprimez les cookies mémorisés par le navigateur.

Déconnexion

La commande de déconnexion

Voici la méthode concernée dans le `trait AuthenticatesUsers` :

Copyright © 2016 Eyrolles.

```
<?php
/**
 * Log the user out of the application.
 *
 * @return \Illuminate\Http\Response
 */
public function getLogout()
{
    return $this->logout();
}

/**
 * Log the user out of the application.
 *
 * @return \Illuminate\Http\Response
 */
public function logout()
{
  Auth::guard($this->getGuard())->logout();
  return redirect(property_exists($this, 'redirectAfterLogout') ? $this-
>redirectAfterLogout : '/');
}
```

L'utilisateur est déconnecté avec la méthode `logout` et il est ensuite redirigé à la route définie par la propriété `redirectAfterLogout` si elle existe, sinon vers la racine `/`.

L'oubli du mot de passe

Il existe une table `password_resets` dans notre base.

#	Nom	Type
1	email	varchar(255)
2	token	varchar(255)
3	created_at	timestamp

La table password_resets

On mémorisera l'adresse électronique, le jeton (`token`) et le `timestamp` (par défaut, les jetons sont valables pendant une heure).

On a aussi un contrôleur.

Copyright © 2016 Eyrolles.

Le contrôleur PasswordController

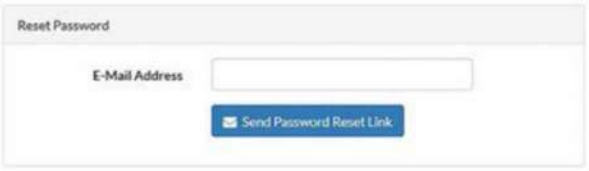
Les URL auront la forme *.../password/...*

Dans le formulaire de connexion, il est prévu un lien en cas d'oubli du mot de passe.

Le lien en cas d'oubli du mot de passe

L'URL correspondante est *.../password/email*.

Voici l'aspect du formulaire.



Le formulaire en cas d'oubli du mot de passe

On demande à l'utilisateur de saisir son adresse électronique pour pouvoir le retrouver dans la table des utilisateurs.

Voici le code dans le `trait ResetsPasswords` :

```
<?php
/**
 * Display the form to request a password reset link.
 *
 * @return \Illuminate\Http\Response
 */
public function getEmail()
{
 return $this->showLinkRequestForm();
}
```

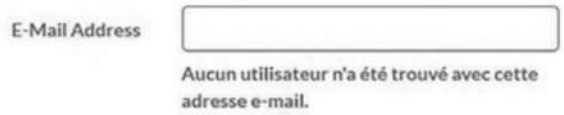
Copyright © 2016 Eyrolles.

```
/**
 * Display the form to request a password reset link.
 *
 * @return \Illuminate\Http\Response
 */
public function showLinkRequestForm()
{
        if (property_exists($this, 'linkRequestView')) {
            return view($this->linkRequestView);
        }

        if (view()->exists('auth.passwords.email')) {
            return view('auth.passwords.email');
        }


  return view('auth.password');
}

/**
 * Send a reset link to the given user.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function postEmail(Request $request)
{
  return $this->sendResetLinkEmail($request);
}

/**
 * Send a reset link to the given user.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function sendResetLinkEmail(Request $request)
{
  $this->validate($request, ['email'=>'required|email']);
  $broker=$this->getBroker();
  $response=Password::broker($broker)->sendResetLink($request-
>only('email'), function (Message $message) {
    $message->subject($this->getEmailSubject());
  });

  switch ($response) {
    case Password::RESET_LINK_SENT:
      return $this->getSendResetLinkEmailSuccessResponse($response);
    case Password::INVALID_USER:
    default:
      return $this->getSendResetLinkEmailFailureResponse($response);
  }
}
```

La soumission du formulaire est prévue dans la méthode `postEmail`. On rencontre deux cas.

Copyright © 2016 Eyrolles.

- L'utilisateur n'est pas valide (l'adresse électronique n'existe pas) : on redirige sur le formulaire avec le message d'erreur dans la variable `error`.



Le message d'erreur pour une adresse inconnue

- L'utilisateur est valide (on lui a envoyé un courriel) : on redirige sur le formulaire avec le message dans la variable `status`.

Le courriel a bien été envoyé.

Qu'est-ce que la méthode trans ?

C'est un *helper* qui permet d'adapter le texte linguistiquement à partir de la valeur de la clé locale dans `config/app.php`. Nous verrons cela en détail dans le chapitre sur la localisation.

Pour que le courriel soit effectivement envoyé, il faut que tout soit bien configuré dans le fichier `.env` :

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.free.fr
MAIL_PORT=25
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=""
```

Et dans `config/mail.php` :

```
'from'=>array('address'=>'moi@free.fr', 'name'=>'Administrateur'),
```

Cette configuration est à adapter selon vos besoins.

La vue pour le courriel

Copyright © 2016 Eyrolles.

En voici le code :

```
Click here to reset your password: {{ url('password/reset/'.$token) }}
```

C'est clair et concis. Voici ce qu'on obtient en réception :

```
Click here to reset your password: http://.../password/
reset/6e8ad7609e1b7cc5a483d81524ca8443f0138b73
```

Voyons ce qui s'est passé dans la table `password_resets`.

email	token	created_at
durand@gem.fr	7ad478ecdcd696c4ebfa649f5f9454f92606220a	2015-01-28 17:34:14

La table password_resets

L'adresse électronique, le jeton (`token`) et le moment de la création sont enregistrés.

Si on utilise l'URL de l'e-mail, on est dirigé sur la route `password/reset` et donc sur la méthode `showResetForm` du `trait ResetsPasswords` :

```
/**
 * Display the password reset view for the given token.
 *
 * If no token is present, display the link request form.
 *
 * @param \Illuminate\Http\Request $request
 * @param string|null $token
 * @return \Illuminate\Http\Response
 */
public function showResetForm(Request $request, $token=null)
{
  if (is_null($token)) {
    return $this->getEmail();
  }
  $email=$request->input('email');
  if (property_exists($this, 'resetView')) {
 return view($this->resetView)->with(compact('token', 'email'));
  }
  if (view()->exists('auth.passwords.reset')) {
    return view('auth.passwords.reset')->with(compact('token', 'email'));
  }
  return view('auth.reset')->with('token', $token);
}
```

Le jeton est transmis dans la variable `$token`. On vérifie sa présence, sinon on renvoie le formulaire. S'il est présent, on retourne le formulaire de saisie du nouveau mot de passe avec la vue `reset` dans le dossier `auth/passwords`.

Copyright © 2016 Eyrolles.

La vue reset

Le formulaire pour le nouveau mot de passe

À la soumission, on tombe sur la méthode `postReset` du `trait` :

```
<?php
/**
 * Reset the given user's password.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function postReset(Request $request)
{
  return $this->reset($request);
}

/**
 * Reset the given user's password.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function reset(Request $request)
{
  $this->validate(
        $request,
       $this->getResetValidationRules(),
        $this->getResetValidationMessages(),
       $this->getResetValidationCustomAttributes()
```

Copyright © 2016 Eyrolles.

```
    );
  $credentials=$request->only(
    'email', 'password', 'password_confirmation', 'token'
  );
  $broker=$this->getBroker();
  $response=Password::broker($broker)->reset($credentials, function ($user,
$password) {
    $this->resetPassword($user, $password);
  });
  switch ($response) {
    case Password::PASSWORD_RESET:
      return $this->getResetSuccessResponse($response);
    default:
      return $this->getResetFailureResponse($request, $response);
  }
}
```

On trouve une validation pour les entrées. Si tout se passe bien, le nouveau mot de passe est mémorisé et l'utilisateur connecté. La redirection dépendra évidemment du succès ou de l'échec.

En résumé

- L'authentification est totalement et simplement prise en charge par Laravel.
- Un *middleware* permet d'effectuer un traitement à l'arrivée ou au départ de la requête.
- On peut utiliser les *middlewares* `Authenticate` et `RedirectIfAuthenticated` pour autoriser ou interdire un accès.
- Un système complet de renouvellement du mot de passe est prévu.

Copyright © 2016 Eyrolles.

Copyright © 2016 Eyro les.

14 La relation 1:n

Pour le moment, nous n'avons manipulé qu'une table avec `Eloquent`. Dans le présent chapitre, nous allons en utiliser deux et les mettre en relation. La relation la plus répandue et la plus simple est celle qui fait correspondre un enregistrement d'une table à plusieurs enregistrements de l'autre table ; on parle de relation de un à plusieurs ou encore de relation de type 1:n. Nous verrons également dans ce chapitre comment créer un *middleware*.

Exemple d'un blog personnel

Comme exemple pour ce chapitre, je vais prendre le cas d'un petit blog personnel avec :

- un affichage des articles ;
- des visiteurs qui pourront consulter les articles ;
- des utilisateurs enregistrés qui pourront aussi rédiger des articles (donc possibilité de se connecter et se déconnecter) ;
- des administrateurs qui auront aussi le droit de supprimer des articles.

Pour ne pas trop alourdir le code, je ne prévoierai pas la modification des articles.

Les données

Migrations

Continuons à utiliser la table `users` des chapitres précédents. Nous allons ajouter une nouvelle table `posts` destinée à mémoriser les articles. Si vous avez déjà créé la table `users` avec des enregistrements, supprimez-la ; nous allons la recréer.

Nous avons déjà défini la migration de la table `users` et vous devez avoir le fichier dans le dossier `app/database/migrations`. Je vous en rappelle le code :

Copyright © 2016 Eyrolles.

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateUsersTable extends Migration
{
  public function up()
  {
    Schema::create('users', function(Blueprint $table)
    {
      $table->increments('id');
      $table->string('name')->unique();
      $table->string('email')->unique();
      $table->string('password', 60);
      $table->boolean('admin')->default(false);
      $table->rememberToken();
      $table->timestamps();
    });
  }

  public function down()
  {
    Schema::drop('users');
  }
}
```

Créons une migration pour la table `posts` :

```
php artisan make:migration create_posts_table
```

Et complétons le code comme suit :

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreatePostsTable extends Migration
{
  public function up()
  {
   Schema::create('posts', function(Blueprint $table) {
      $table->increments('id');
      $table->timestamps();
      $table->string('titre', 80);
      $table->text('contenu');
      $table->integer('user_id')->unsigned();
      $table->foreign('user_id')
            ->references('id')
             ->on('users')
             ->onDelete('restrict')
            ->onUpdate('restrict');
    });
 }
```

Copyright © 2016 Eyrolles.

```
    public function down()
    {
        Schema::table('posts', function(Blueprint $table) {
            $table->dropForeign('posts_user_id_foreign');
        });
        Schema::drop('posts');
    }
}
```

Normalement, vous devez avoir trois migrations.

Les trois migrations

Lancez la migration :

```
php artisan migrate
```

Vous devez ainsi vous retrouver avec les trois tables dans votre base, ainsi que la table `migrations`.

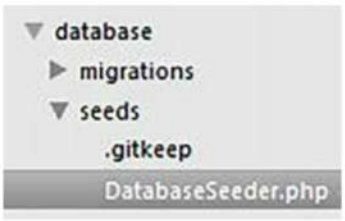
Les quatre tables

Pour que la population que nous allons créer ensuite fonctionne, il faut repartir sur une nouvelle migration pour la table `users`. En effet, on va avoir besoin que la clé de la table commence à 1.

Copyright © 2016 Eyrolles.

Population

Nous allons remplir nos tables avec des enregistrements pour réaliser nos essais. Créons pour cela deux fichiers dans le dossier `database/seeds`. Normalement, vous devez déjà avoir dans ce dossier le fichier `DatabaseSeeder.php`.



Le fichier DatabaseSeeder

En voici le code :

```
<?php
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
  /**
   * Run the database seeds.
   *
   * @return void
   */
  public function run()
  {
    // $this->call(UserTableSeeder::class);
  }
}
```

La méthode `run` est destinée à exécuter les fichiers pour la population. Vous avez déjà la ligne commentée de lancement pour la table `users`. Nous allons la dé-commenter et ajouter le code pour la table `posts` :

```
<?php
public function run()
{
  $this->call(UserTableSeeder::class);
  $this->call(PostTableSeeder::class);
}
```

Placez bien les lignes dans cet ordre ; vous comprendrez bientôt pourquoi c'est nécessaire.

Ensuite, on va créer le fichier `UserTableSeeder.php` pour la population de la table `users` :

Copyright © 2016 Eyrolles.

```
<?php
use Illuminate\Database\Seeder;

class UserTableSeeder extends Seeder
{
  public function run()
  {
    DB::table('users')->delete();
    for($i=0; $i<10; ++$i)
    {
      DB::table('users')->insert([
        'name'=>'Nom' . $i,
        'email'=>'email' . $i . '@blop.fr',
        'password'=>bcrypt('password' . $i),
        'admin'=>rand(0,1)
      ]);
    }
  }
}
```

Cela crée dix utilisateurs.

On prévoit aussi le fichier `PostTableSeeder.php` avec le code suivant :

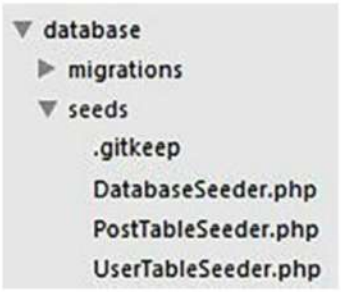
```
<?php
use Illuminate\Database\Seeder;
use Carbon\Carbon;

class PostTableSeeder extends Seeder {
  private function randDate()
  {
    return Carbon::createFromDate(null, rand(1,12), rand(1,28));
  }

  public function run()
  {
    DB::table('posts')->delete();
    for($i=0; $i<100; ++$i)
    {
      $date=$this->randDate();
      DB::table('posts')->insert([
        'titre'=>'Titre' . $i,
        'contenu'=>'Contenu' . $i . 'Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore
magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco
laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor
in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla
pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui
officia deserunt mollit anim id est laborum.',
        'user_id'=>rand(1,10),
        'created_at'=>$date,
        'updated_at'=>$date
      ]);
    }
  }
}
```

Copyright © 2016 Eyrolles.

Cela générera cent articles affectés de façon aléatoire aux dix utilisateurs. Nous étudierons bientôt comment s'effectue la liaison entre les deux.

La classe `Carbon` (*http://carbon.nesbot.com/docs/*), contenue dans un package chargé par Laravel, facilite la manipulation des dates. N'hésitez pas à l'utiliser dès que vous avez des dates à gérer.

Vous devez maintenant avoir les fichiers suivants dans le dossier des populations.



Les fichiers de population

Il suffit d'utiliser la commande d'`Artisan` pour lancer la population :

```
php artisan db:seed
```

Normalement, vous devez obtenir les deux tables remplies à l'issue de cette commande.

Si vous recevez un message vous disant que la classe `userTableSeeder` ou l'autre classe n'est pas trouvée, exécutez la commande `composer dumpautoload`, puis relancez la population.

id	name	email	password	admin
1	Nom0	email0@blop.fr	$2y$10$ApuwKOZdqeY1ZCi0YPHpeuTDaecarM93Vk1NNAeREMP...	1
2	Nom1	email1@blop.fr	$2y$10$yZN9Z1uw2iSh6/6Ci3.CJO8IFfpe6EQUtk5uo9IiQIR...	1
3	Nom2	email2@blop.fr	$2y$10$k7xHtOVBFUCv5Ei.Ju7h.OSLyqaht/iRCef.LJOwGT...	0
4	Nom3	email3@blop.fr	$2y$10$WL7UwNttEDLNFX2b.i9Qn.Ow01ZT3sgtRdwyY.y7gVl...	1
5	Nom4	email4@blop.fr	$2y$10$5r2Bp8HFtKCxB9fpR/ogCeWunTtempqdoZm6ONwktS...	0
6	Nom5	email5@blop.fr	$2y$10$PXYviimun0D1Noygfka/kONsgpUrfiO.SydEyji3qvz...	0
7	Nom6	email6@blop.fr	$2y$10$ubXPT/Wh9f65rCHU5i./Lehsq51OZTJju8rFn1OW4Pe...	0
8	Nom7	email7@blop.fr	$2y$10$jsurpKAc3qfLBZ7ycbztO.0nbS9xd8/miuQl6R8wDmx...	0
9	Nom8	email8@blop.fr	$2y$10$9EF.gS8x53nzwHLbGp5Rf.mX7ckTsKiVLnJ11LrqGFP...	1
10	Nom9	email9@blop.fr	$2y$10$a84ghb9xhXiVEHjfUl5XZe5OiNbtRJfjBRz5N0Lk.xj...	1

La table users

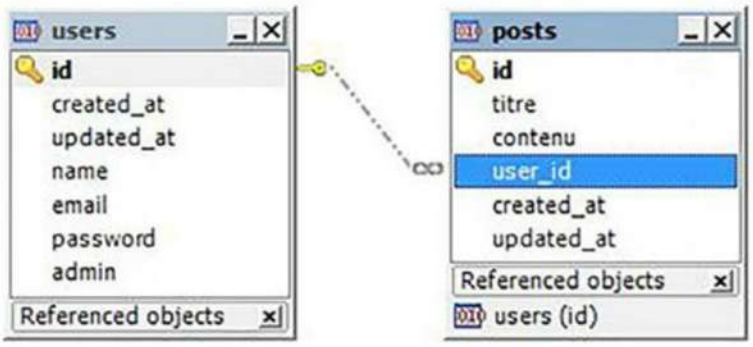
Copyright © 2016 Eyrolles.

La relation

On a la situation suivante :

- un utilisateur peut écrire plusieurs articles ;
- un article est écrit par un seul utilisateur.

Il faut trouver un moyen de référencer cette relation dans les tables. Le principe est simple : on prévoit dans la table `posts` une ligne destinée à recevoir l'identifiant de l'utilisateur rédacteur de l'article. On appelle cette ligne une clé étrangère parce qu'on enregistre ici la clé d'une autre table. Voici une représentation visuelle de cette relation.



La clé étrangère

Vous voyez la relation dessinée entre la clé `id` dans la table `users` et la clé étrangère `user_id` dans la table `posts`. La migration qu'on a créée informe la base de cette relation, via le code suivant :

```
<?php
$table->foreign('user_id')
      ->references('id')
      ->on('users')
      ->onDelete('restrict')
      ->onUpdate('restrict');
```

Dans la table, on déclare une clé étrangère (`foreign`) nommée `user_id` qui référence (`references`) la ligne `id` dans la table (`on`) `users`. En cas de suppression (`onDelete`) ou de modification (`onUpdate`), on a une restriction (`restrict`). Que signifient ces deux dernières conditions ?

Imaginez que vous ayez un utilisateur avec l'`id 5` associé à deux articles. Dans la table `posts`, deux enregistrements ont donc un `user_id` de valeur `5`. Si on supprime l'utilisateur, la clé étrangère de ces deux enregistrements ne correspond plus à aucun enregistrement dans la table `users`. En indiquant `restrict`, on empêche la suppression d'un utilisateur auquel est associé au moins un article. Il faut commencer par supprimer ses articles avant de le supprimer lui-même. On dit que la base assure

Copyright © 2016 Eyrolles.

l'*intégrité référentielle*. Elle n'accepte pas non plus qu'on utilise pour `user_id` une valeur qui n'existe pas dans la table `users`.

Une autre possibilité est `cascade` à la place de `restrict`. Dans ce cas, si vous supprimez un utilisateur, tous les articles associés sont également effacés. C'est une option qui est rarement utilisée parce qu'elle peut s'avérer dangereuse, surtout dans une base comportant de multiples tables en relation. Néanmoins, c'est aussi une stratégie très efficace parce que c'est le moteur de la base de données qui se charge de gérer les enregistrements en relation ; vous n'avez ainsi pas à vous en soucier au niveau du code.

On pourrait aussi ne pas signaler à la base qu'il existe une relation et la gérer seulement dans notre code. Cependant, c'est encore plus dangereux parce que la moindre erreur de gestion des enregistrements dans votre code risque d'avoir des conséquences importantes dans votre base avec de multiples incohérences.

Les modèles

Nous avons déjà un modèle `User` (`app/User.php`). Il faudra juste lui ajouter une méthode pour trouver facilement les articles d'un utilisateur :

```
<?php
public function posts()
{
  return $this->hasMany('App\Post');
}
```

On déclare ici qu'un utilisateur a plusieurs (`hasMany`) articles (`posts`). On aura ainsi une méthode pratique pour récupérer les articles d'un utilisateur.

Soyez vigilant avec les espaces de noms !

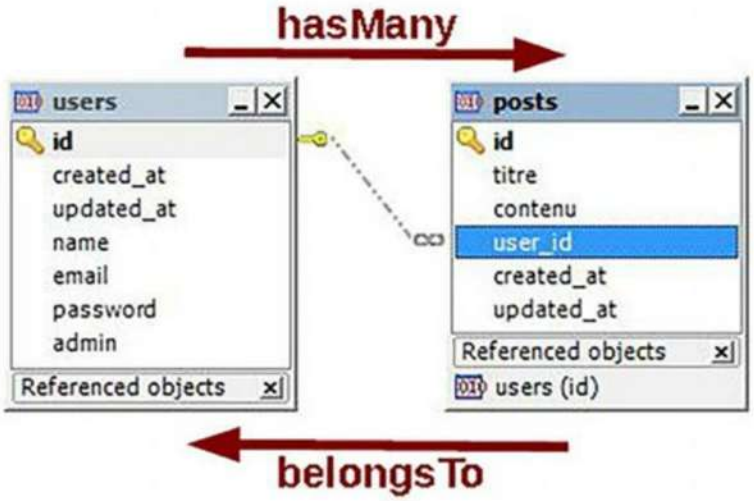
Il nous faut aussi le modèle `Post` :

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
  protected $fillable=['titre','contenu','user_id'];

  public function user()
  {
    return $this->belongsTo('App\User');
  }
}
```

Copyright © 2016 Eyrolles.

Ici, la méthode `user` (au singulier) trouve l'utilisateur auquel appartient (`belongsTo`) l'article. C'est donc la réciproque de la méthode précédente.

Voici une schématisation de cette relation avec les deux méthodes.



Les deux méthodes de la relation

Je vous rappelle que si vous ne spécifiez pas de manière explicite le nom de la table dans un modèle, Laravel le déduit à partir du nom du modèle qu'il met au pluriel (à la mode anglaise) et avec la première lettre en minuscule.

Les deux méthodes mises en place récupèrent facilement un enregistrement lié. Par exemple, cherchons tous les articles de l'utilisateur qui a l'`id 1` :

```
<?php
$articles=App\User::find(1)->posts;
```

De la même manière, on peut trouver l'utilisateur qui a écrit l'article d'`id 1` :

```
<?php
$user=App\Post::find(1)->user;
```

Vous voyez que le codage devient limpide avec ces méthodes !

Laravel dispose de l'outil `tinker` qui permet d'entrer des commandes dans la console et ainsi d'interagir directement avec l'application. Il faut le démarrer avec la commande `php artisan tinker`. On peut ensuite l'utiliser directement :

Copyright © 2016 Eyrolles.

```
php artisan tinker
Psy Shell v0.6.1 (PHP 5.5.12 ÔÇö cli) by Justin Hileman
>>> App\User::find(1)->posts =>Illuminate\Database\Eloquent\Collection
{#661 all: [App\Post {#662 id:47,
created_at: "2015-03-19 12:39:12",
updated_at: "2015-03-19 12:39:12",
titre: "Titre46",
contenu: "Contenu46 Lorem ipsum dolor sit amet, consectetur adipisicing
elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint
occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit
anim id est laborum.",
user_id: 1,}, ...
```

Contrôleur et routes

Contrôleur

Tout est en place au niveau des données ; voyons donc un peu comment gérer tout cela. Créons le contrôleur pour les articles, `PostController`. Il doit gérer plusieurs choses :

- recevoir la requête pour afficher les articles du blog et fournir la réponse adaptée ;
- recevoir la requête pour le formulaire destiné à la création d'un nouvel article et y répondre ;
- recevoir le formulaire soumis (réservé à un utilisateur connecté) et l'enregistrer ;
- recevoir la demande de suppression d'un article (réservé à un administrateur) et supprimer l'enregistrement correspondant.

Pour simplifier, je ne vais pas prévoir la possibilité de modifier un article.

J'utiliserai un contrôleur de ressource. Voici son code :

```
<?php
namespace App\Http\Controllers;

use App\Repositories\PostRepository;
use App\Http\Requests\PostRequest;

class PostController extends Controller
{
  protected $postRepository;
  protected $nbrPerPage=4;

 public function __construct(PostRepository $postRepository)
  {
    $this->middleware('auth', ['except'=>'index']);
   $this->middleware('admin', ['only'=>'destroy']);
```

Copyright © 2016 Eyrolles.

```
    $this->postRepository=$postRepository;
  }

  public function index()
  {
    $posts=$this->postRepository->getPaginate($this->nbrPerPage);
    $links=$posts->render();
    return view('posts.liste', compact('posts', 'links'));
  }

  public function create()
  {
    return view('posts.add');
  }

 public function store(PostRequest $request)
  {
    $inputs=array_merge($request->all(), ['user_id'=>$request->user()->id]);
    $this->postRepository->store($inputs);
    return redirect(route('post.index'));
  }

  public function destroy($id)
  {
    $this->postRepository->destroy($id);
    return redirect()->back();
  }
}
```

Comme à l'accoutumée, j'injecte la requête de formulaire et le gestionnaire.

Notez l'utilisation des *middlewares* pour filtrer les utilisateurs ; nous y reviendrons un peu plus loin.

Routes

On a vu dans le chapitre sur les ressources comment créer les routes de ce genre de contrôleur. Il faut juste indiquer qu'on ne veut pas utiliser les sept méthodes disponibles, mais seulement certaines :

```
<?php
Route::resource('post', 'PostController', ['except'=>['show', 'edit',
'update']]);
  ...
```

Avec `except`, j'indique que je ne veux pas de route pour les trois méthodes citées. Voici ce que cela donne en utilisant `Artisan` pour visualiser les routes (`php artisan route:list`).

Copyright © 2016 Eyrolles.

```
+--------------+-----------------------------------------------+
| Name         | Action                                        |
+--------------+-----------------------------------------------+
| post.index   | App\Http\Controllers\PostController@index     |
| post.create  | App\Http\Controllers\PostController@create    |
| post.store   | App\Http\Controllers\PostController@store     |
| post.destroy | App\Http\Controllers\PostController@destroy   |
```

Les routes

Le gestionnaire des articles

Nous plaçons les fichiers dans le dossier `app/Repositories`, comme nous l'avons déjà fait pour la gestion des utilisateurs.

Le gestionnaire des articles

```php
<?php
namespace App\Repositories;

use App\Post;

class PostRepository
{
  protected $post;

  public function __construct(Post $post)
  {
    $this->post=$post;
  }

 public function getPaginate($n)
  {
    return $this->post->with('user')
    ->orderBy('posts.created_at', 'desc')
    ->paginate($n);
  }

 public function store($inputs)
  {
    $this->post->create($inputs);
  }

 public function destroy($id)
  {

    $this->post->findOrFail($id)->delete();
  }
}
```

Nous discuterons plus loin de l'utilité de toutes ces méthodes.

Copyright © 2016 Eyrolles.

Les middlewares

Dans le contrôleur, on applique deux *middlewares* :

- `auth` : accès réservé aux utilisateurs authentifiés, à part pour la méthode `index` pour afficher le blog ;
- `admin` : accès réservé aux administrateurs pour la méthode `destroy`.

Le premier est déjà prévu dans Laravel, mais le second n'existe pas. Pour le créer, nous utilisons `Artisan` :

```
php artisan make:middleware Admin
```

Le middleware pour l'administration

On obtient le code suivant, auquel nous ajoutons nos instructions :

```
<?php
namespace App\Http\Middleware;

use Closure;

class Admin
{
  /**
   * Handle an incoming request.
   *
   * @param \Illuminate\Http\Request $request
   * @param \Closure $next
   * @return mixed
   */
  public function handle($request, Closure $next)
  {
    if ($request->user()->admin)
    {
      return $next($request);
    }
    return new RedirectResponse(url('post/liste'));
  }
}
```

Si l'utilisateur n'est pas un administrateur, on redirige sur l'affichage du blog. Remarquez qu'on ne vérifie pas à ce niveau qu'on a un utilisateur authentifié, parce que dans le constructeur du contrôleur, le filtre `auth` est placé avant le filtre `admin`. Si c'était

Copyright © 2016 Eyrolles.

l'inverse, on tomberait évidemment sur une erreur en cas de tentative d'accès à l'URL pour la suppression d'un article.

On a créé le *middleware*, mais cela ne suffit pas ; il faut maintenant un lien entre le nom qu'on veut donner au filtre et la classe qu'on vient de créer. Regardez dans le fichier `app/Http/Kernel.php` qui contient tous les *middlewares* déclarés et ajoutez la ligne de code correspondant à `admin` :

```
<?php
protected $routeMiddleware=[
  'auth'=>\App\Http\Middleware\Authenticate::class,
  'auth.basic'=>\Illuminate\Auth\Middleware\
AuthenticateWithBasicAuth::class,
  'guest'=>\App\Http\Middleware\RedirectIfAuthenticated::class,
  'throttle'=>\Illuminate\Routing\Middleware\ThrottleRequests::class,
  'admin'=>\App\Http\Middleware\Admin::class,
];
```

La validation

Voyons la validation. Il faut encore créer une requête de formulaire :

```
php artisan make:request PostRequest
```

Elle se place dans le dossier suivant.

La requête de formulaire

On complète le code :

```
<?php
namespace App\Http\Requests;

use App\Http\Requests\Request;

class PostRequest extends Request
{
  public function authorize()
  {
    return true;
  }

  public function rules()
  {
```

Copyright © 2016 Eyrolles.

```
    return [
      'titre'=>'required|max:80',
      'contenu'=>'required'
    ];
  }
}
```

Le fonctionnement

Liste des articles

La liste des articles est obtenue avec l'URL *.../post* (verbe `get`) qui arrive sur la méthode `index` du contrôleur :

```
<?php
public function index()
{
  $posts=$this->postRepository->getPaginate($this->nbrPerPage);
  $links=$posts->render();
  return view('posts.liste', compact('posts', 'links'));
}
```

Ici, on envoie le nombre d'articles par page (placé dans la propriété `$nbrPerPage`) à la méthode `getPaginate` du gestionnaire :

```
<?php
public function getPaginate($n)
{
  return $this->post->with('user')
  ->orderBy('posts.created_at', 'desc')
  ->paginate($n);
}
```

On veut les articles avec (`with`) l'utilisateur (`user`), dans l'ordre des dates de création (`posts.created_at`) descendant (`desc`) avec une pagination de `n` articles (`$n`).

Il existe la méthode `latest` (et `oldest` pour l'inverse) qui simplifie la syntaxe :

```
<?php
return $this->post->with('user')
->latest('posts.created_at')
->paginate($n);
```

Copyright © 2016 Eyrolles.

Ajout d'un article

La demande du formulaire de création d'un article s'effectue avec l'URL *.../post/create* (verbe `get`). Le contrôleur renvoie directement la vue :

```
<?php
public function create()
{
    return view('posts.add');
}
```

Le retour du formulaire se fait avec l'URL *.../post* (verbe `post`).

On arrive sur la méthode `store` du contrôleur :

```
<?php
public function store(PostRequest $request)
{
  $inputs=array_merge($request->all(), ['user_id'=>$request->user()->id]);
  $this->postRepository->store($inputs);
  return redirect(route('post.index'));
}
```

On injecte la requête de formulaire et on récupère les informations sur le titre et le contenu. On sait que l'utilisateur est forcément connecté, alors on récupère son identifiant avec la requête. Si la validation se passe bien, on envoie à la méthode `store` du repository :

```
<?php
public function store($inputs)
{
  $this->post->create($inputs);
}
```

Suppression d'un article

On supprime un article avec l'URL *.../post/id* (verbe `delete`), où `id` représente l'identifiant de l'article à supprimer. On tombe sur la méthode `destroy` du contrôleur :

```
<?php
public function destroy($id)
{
  $this->postRepository->destroy($id);
  return redirect()->back();
}
```

Cette dernière envoie à la méthode `destroy` du repository :

Copyright © 2016 Eyrolles.

```php
<?php
public function destroy($id)
{
  $this->post->findOrFail($id)->delete();
}
```

Là, on supprime l'article avec la méthode `delete` du modèle.

Les vues

Template

On va un peu modifier notre template (`resources/views/template.blade.php`) :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Mon joli site</title>
    {!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/
bootstrap.min.css') !!}
    {!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/
bootstrap-theme.min.css') !!}
    <!--[if lt IE 9]>
      {{ Html::style('https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.
js') }}
      {{ Html::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.
min.js') }}
    <![endif]-->
    <style>textarea{resize:none;}</style>
  </head>
  <body>
    <header class="jumbotron">
      <div class="container">
        <h1 class="page-header">{!! link_to_route('post.index', 'Mon joli
blog') !!}</h1>
        @yield('header')
      </div>
    </header>
    <div class="container">
      @yield('contenu')
    </div>
  </body>
</html>
```

Copyright © 2016 Eyrolles.

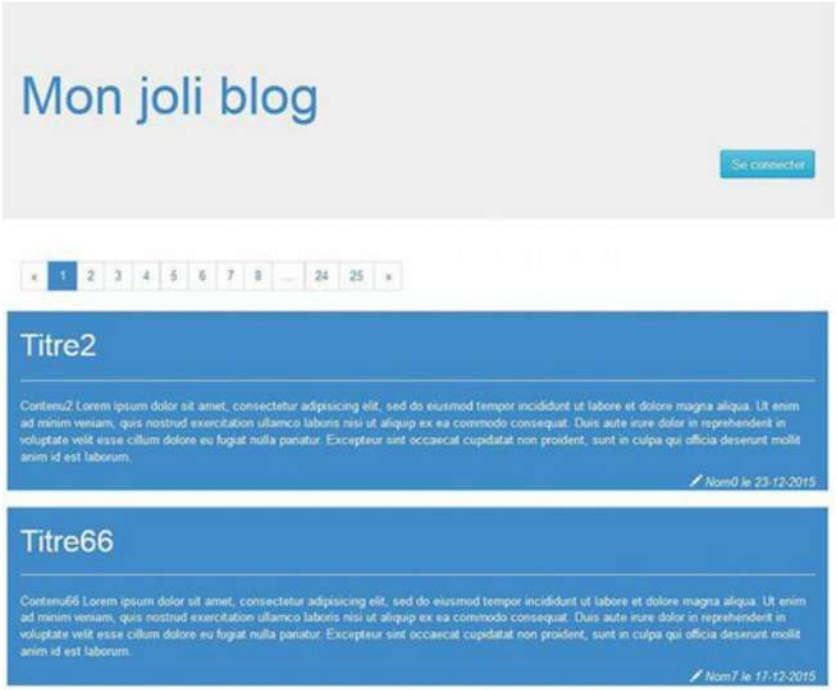
Vue pour afficher les articles

Nous avons besoin d'une vue pour afficher les articles du blog et quelques boutons pour la gestion (`resources/views/posts/liste.blade.php`) :

```
@extends('template')

@section('header')
  @if(Auth::check())
    <div class="btn-group pull-right">
      {!! link_to_route('post.create', 'Créer un article', [],
['class'=>'btn btn-info']) !!}
      {!! link_to('logout', 'Deconnexion', ['class'=>'btn btn-warning']) !!}
    </div>
  @else
    {!! link_to('login', 'Se connecter', ['class'=>'btn btn-info pull-
right']) !!}
  @endif
@endsection

@section('contenu')
  @if(isset($info))
    <div class="row alert alert-info">{{$info}}</div>
  @endif
  {!!$links!!}
  @foreach($posts as $post)
    <article class="row bg-primary">
      <div class="col-md-12">
        <header>
          <h1>{{$post->titre}}</h1>
        </header>
        <hr>
        <section>
          <p>{{$post->contenu}}</p>
          @if(Auth::check() and Auth::user()->admin)
            {!! Form::open(['method'=>'DELETE', 'route'=>['post.destroy',
$post->id]]) !!}
              {!! Form::submit('Supprimer cet article', ['class'=>'btn
btn-danger btn-xs', 'onclick'=>'return confirm(\'Vraiment supprimer cet
article ?\')']) !!}
            {!! Form::close() !!}
          @endif
          <em class="pull-right">
            <span class="glyphicon glyphicon-pencil"></span> {{$ post->user-
>name }} le {!! $post->created_at->format('d-m-Y') !!}
          </em>
        </section>
      </div>
    </article>
    <br>
  @endforeach
  {!!$links!!}
@endsection
```

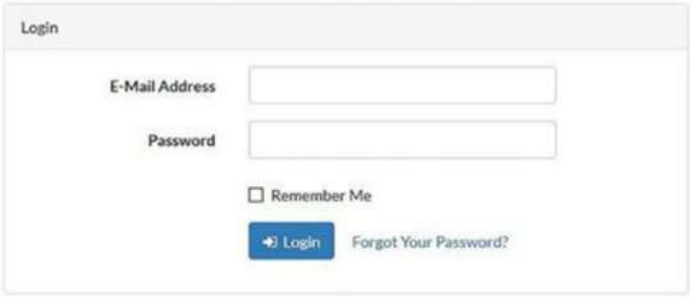
Voici l'aspect obtenu pour un utilisateur non connecté.

Copyright © 2016 Eyrolles.



L'aspect du blog

Le bouton ***Se connecter*** envoie sur l'URL *.../login*, ce qui doit aboutir sur le formulaire de connexion si vous avez tout en place comme nous l'avons prévu précédemment.



Le formulaire de connexion

Pour mémoire, j'utilise ici les vues par défaut de Laravel. Vous êtes évidemment libres de franciser ces vues ou, encore mieux, les adapter au langage de l'utilisateur (nous verrons cet aspect dans un chapitre ultérieur).

Pour que votre application fonctionne bien avec le `AuthController`, il faut modifier la propriété `redirectTo` dans ce contrôleur :

```
<?php
protected $redirectTo='post';
```

Ainsi, vous serez bien redirigé vers le blog en cas de connexion.

Copyright © 2016 Eyrolles.

De la même manière, il faut prévoir une redirection après déconnexion dans le même contrôleur :

```
<?php
protected $redirectAfterLogout='post';
```

La génération des articles dans la vue se fait avec un `foreach` :

```
@foreach($posts as $post)
   ...
@endforeach
```

Si vous vous connectez avec un utilisateur qui n'est pas administrateur (regardez dans votre table pour en trouver un car, comme la population est aléatoire, on ne sait pas à l'avance qui l'est et qui ne l'est pas). Vous retournez au blog avec deux boutons supplémentaires.

Les boutons pour un utilisateur de base

On utilise une condition pour adapter la page :

```
@if(Auth::check())
    ...
@else
    ...
@endif
```

La méthode `check` de la classe `Auth` permet de savoir si l'utilisateur est connecté.

Il existe l'*helper* `auth()` qui vous évite le recours à la façade :
`@if(auth()->check())`

Le premier bouton, ***Créer un article***, génère l'URL *.../post/create*, ce qui a pour effet d'obtenir le formulaire de création.

Le second bouton, ***Déconnexion***, génère l'URL *.../logout*, qui correspond aussi à ce que nous avons vu dans le chapitre précédent.

Si l'utilisateur connecté est un administrateur, alors il a en plus pour chaque article un bouton de suppression.

Copyright © 2016 Eyrolles.

Le bouton de suppression

On utilise encore une condition pour détecter un administrateur :

```
@if(Auth::check() and Auth::user()->admin)
  ...
@endif
```

Il faut que l'utilisateur soit connecté (`check`) et qu'il soit un administrateur (`user()->admin`).

J'ai prévu une confirmation de la suppression avec un peu de JavaScript.

Vue pour créer un article

Voici maintenant la vue pour le formulaire de création d'un article (`resources/views/posts/add.blade.php`) :

```
@extends('template')

@section('contenu')
  <br>
  <div class="col-sm-offset-3 col-sm-6">
    <div class="panel panel-info">
      <div class="panel-heading">Ajout d'un article</div>
      <div class="panel-body">
        {!! Form::open(['route'=>'post.store']) !!}
          <div class="form-group {!! $errors->has('titre') ? 'has-
error' : '' !!}">
            {!! Form::text('titre', null, ['class'=>'form-control',
'placeholder'=>'Titre']) !!}
            {!! $errors->first('titre', '<small class="help-block">:message</
small>') !!}
          </div>
          <div class="form-group {!! $errors->has('contenu') ? 'has-
error' : '' !!}">
            {!! Form::textarea ('contenu', null, ['class'=>'form-control',
'placeholder'=>'Contenu']) !!}
```

Copyright © 2016 Eyrolles.

```
            {!! $errors->first('contenu', '<small class="help-
block">:message</small>') !!}
          </div>
          {!! Form::submit('Envoyer !', ['class'=>'btn btn-info pull-
right']) !!}
        {!! Form::close() !!}
      </div>
    </div>
    <a href="javascript:history.back()" class="btn btn-primary">
      <span class="glyphicon glyphicon-circle-arrow-left"></span>Retour
    </a>
  </div>
@endsection
```

Il n'y a rien de bien nouveau dans cette vue. Voici son apparence.

Formulaire de création d'un article

On peut entrer le titre et le contenu, qui seront ensuite validés dans le contrôleur et enregistrés si tout se passe bien.

Il arrive parfois que les vues ne se génèrent pas correctement. Laravel utilise un cache pour les vues en `storage/framework/views`. Vous pouvez sans problème supprimer tous les fichiers (mais pas `.gitignore` !) pour obliger Laravel à générer de nouvelles vues. Il existe une commande `Artisan` pour le faire : `php artisan view:clear`.

En résumé

- Une relation de type 1:n nécessite la création d'une clé étrangère côté n.
- On peut remplir les tables d'enregistrements avec la population.
- Une relation dans la base nécessite la mise en place de méthodes spéciales dans les modèles.
- Avec les *middlewares*, il est facile de gérer l'accès aux méthodes des contrôleurs.

Copyright © 2016 Eyrolles.

15 La relation n:n

Dans le précédent chapitre, nous avons décrit la relation de type 1:n, la plus simple et la plus répandue. Nous allons étudier la relation de type n:n, plus délicate à comprendre et à mettre en œuvre. `Eloquent` permet de simplifier la gestion de ce type de relation.

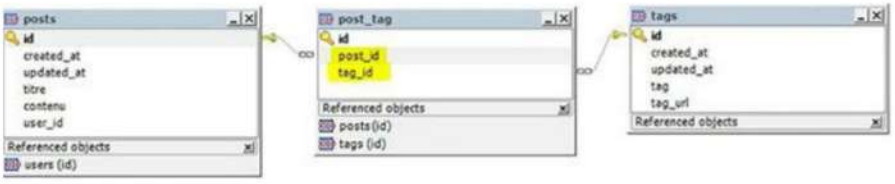
Je vais poursuivre l'exemple du blog personnel débuté au chapitre précédent en ajoutant la possibilité d'associer des mots-clés (`tags`) aux articles. Ce chapitre est un peu long mais j'ai préféré tout rassembler ici.

Les données

Pour comprendre la relation de type n:n, imaginez une relation entre deux tables A et B dans laquelle on peut avoir :

- une ligne de la table A en relation avec plusieurs lignes de la table B ;
- une ligne de la table B en relation avec plusieurs lignes de la table A.

Cette relation ne se résout pas comme nous l'avons vu au chapitre précédent avec une simple clé étrangère dans une des tables. En effet, il nous faudrait des clés dans les deux tables et plusieurs clés, ce qui n'est pas possible à réaliser. La solution consiste à créer une table intermédiaire (nommée table pivot) qui sert à mémoriser les clés étrangères. Voici un schéma de ce que nous allons réaliser.



La table pivot

Copyright © 2016 Eyrolles.

La table pivot `post_tag` contient deux clés :

- `post_id` pour mémoriser la clé de la table `posts` ;
- `tag_id` pour mémoriser la clé de la table `tags`.

De cette façon, on peut avoir plusieurs enregistrements liés entre les deux tables. Évidemment, cela demande un peu plus d'intendance au niveau du code parce qu'il y a une table supplémentaire à gérer.

Par convention, le nom de la table pivot est composé des deux noms des tables au singulier pris dans l'ordre alphabétique.

Migrations

Nous allons continuer à utiliser les tables `users` et `posts` des chapitres précédents. Créons en plus une nouvelle table `tags` destinée à mémoriser les mots-clés. Commencez par supprimer toutes les tables de votre base de données, sinon vous risquez de tomber sur des conflits avec les enregistrements que nous allons créer.

Supprimez aussi la table `migrations`.

Normalement, vous devez déjà disposer des migrations pour les tables `users`, `password_resets` et `posts`. Nous allons ajouter les deux tables `tags` et `post_tag`.

Table tags

Créez une nouvelle migration pour la table `tags` :

```
php artisan make:migration create_tags_table
```

Donnez-lui le code suivant :

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateTagsTable extends Migration
{
  public function up()
  {
    Schema::create('tags', function(Blueprint $table) {
      $table->increments('id');
      $table->timestamps();
      $table->string('tag', 50)->unique();
      $table->string('tag_url', 60)->unique();
```

Copyright © 2016 Eyrolles.

```
    });
  }

  public function down()
  {
    Schema::drop('tags');
  }
}
```

On prévoit les champs :

- `id` : clé unique incrémentée ;
- `created_at` et `updated_at` créés par `timestamps` ;
- `tag` : le mot-clé unique limité à 50 caractères ;
- `tag_url` : la version du mot-clé à inclure dans l'URL (avec 60 comme limite pour couvrir les cas les plus défavorables).

Il nous faut deux champs pour le mot-clé. En effet, il faudra qu'on le transmette dans l'URL pour la recherche par mot-clé. Or, l'utilisateur risque de rentrer des accents par exemple (ou pire des barres obliques `/`) ; nous allons donc convertir ces caractères spéciaux en caractères adaptés aux URL.

Table post_tag

Créez une nouvelle migration pour la table `post_tag` :

```
php artisan make:migration create_post_tag_table
```

Puis donnez-lui le code suivant :

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreatePostTagTable extends Migration
{
  public function up()
  {
    Schema::create('post_tag', function(Blueprint $table) {
      $table->increments('id');
      $table->integer('post_id')->unsigned();
      $table->integer('tag_id')->unsigned();
      $table->foreign('post_id')->references('id')->on('posts')
            ->onDelete('restrict')
            ->onUpdate('restrict');
      $table->foreign('tag_id')->references('id')->on('tags')
            ->onDelete('restrict')
            ->onUpdate('restrict');
    });
  }
```

Copyright © 2016 Eyrolles.

```
  public function down()
  {
    Schema::table('post_tag', function(Blueprint $table) {
      $table->dropForeign('post_tag_post_id_foreign');
      $table->dropForeign('post_tag_tag_id_foreign');
    });
    Schema::drop('post_tag');
  }
}
```

On prévoit les champs :

- `post_id` : clé étrangère pour la table `posts` ;
- `tag_id` : clé étrangère pour la table `tags`.

J'ai encore prévu l'option `restrict` pour les cascades afin de sécuriser les opérations sur la base.

Normalement, vous devez rencontrer les migrations suivantes.

Les migrations

Lancez les migrations :

```
php artisan migrate
```

Vous devez ainsi vous retrouver avec six tables dans votre base.

Les tables

Copyright © 2016 Eyrolles.

Pour ne pas recevoir d'erreur, il faut que les migrations soient lancées dans le bon ordre ! L'ordre des migrations est donné par leur date de création. Il suffit donc de changer la date dans le nom des migrations pour en changer l'ordre.

Population

Vous avez déjà les fichiers pour les tables `users` et `posts`. On créera celui pour les tags (`TagTableSeeder.php`) :

```
<?php
use Illuminate\Database\Seeder;
use Carbon\Carbon;

class TagTableSeeder extends Seeder
{
  private function randDate()
  {
    return Carbon::createFromDate(null, rand(1,12), rand(1,28));
  }

  public function run()
  {
    DB::table('tags')->delete();
    for($i=0; $i<20; ++$i)
    {
      $date=$this->randDate();
      DB::table('tags')->insert(array(
        'tag'=>'tag' . $i,
        'tag_url'=>'tag' . $i,
        'created_at'=>$date,
        'updated_at'=>$date
      ));
    }
  }
}
```

On obtiendra ainsi vingt mots-clés.

On crée aussi le fichier pour la table pivot (`PostTagTableSeeder.php`) :

```
<?php
use Illuminate\Database\Seeder;

class PostTagTableSeeder extends Seeder
{
 public function run()
  {
    for($i=1; $i<=100; ++$i)
   {
      $numbers=range(1,20);
     shuffle($numbers);
     $n=rand(3,6);
```

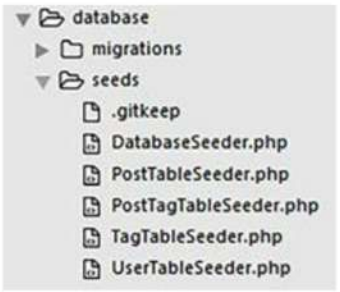
Copyright © 2016 Eyrolles.

```
        for($j=1; $j<$n; ++$j)
        {
          DB::table('post_tag')->insert(array(
            'post_id'=>$i,
            'tag_id'=>$numbers[$j]
          ));
        }
      }
    }
}
```

De façon aléatoire, on crée cent affectations de mots-clés aux articles.
Il ne reste plus qu'à mettre à jour le fichier `DatabaseSeeder.php` :

```
<?php
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
  /**
   * Run the database seeds.
   *
   * @return void
   */
  public function run()
  {
    $this->call(UserTableSeeder::class);
    $this->call(PostTableSeeder::class);
    $this->call(TagTableSeeder::class);
    $this->call(PostTagTableSeeder::class);
  }
}
```

Vous devez obtenir les fichiers suivants.



Les fichiers de population

Il ne reste plus qu'à lancer la population :

```
php artisan db:seed
```

Copyright © 2016 Eyrolles.

Si vous recevez un message vous disant qu'une des classes n'existe pas lancez un `composer dumpautoload`. Si les tables avaient commencé à se remplir, repartez de zéro en les supprimant toutes.

Il est possible de construire en bloc une migration et une population avec la syntaxe : `php artisan migrate --seed`.

Les modèles

On doit déclarer la relation n:n dans le modèle `Post` :

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
  protected $fillable=['titre','contenu','user_id'];

  public function user()
  {
    return $this->belongsTo('App\User');
  }

  public function tags()
  {
    return $this->belongsToMany('App\Tag');
  }
}
```

La méthode `tags` permettra de récupérer les mots-clés qui sont en relation avec l'article. On utilise la méthode `belongsToMany` d'`Eloquent` pour le faire.

On a aussi besoin d'un modèle pour les mots-clés :

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Tag extends Model
{
  protected $fillable=['tag','tag_url'];

  public function posts()
  {
    return $this->belongsToMany('App\Post');
  }
}
```

Copyright © 2016 Eyrolles.

On a la méthode réciproque de la précédente : `posts` permettra de récupérer les articles en relation avec le mot-clé.

Voici une schématisation de cette relation avec les deux méthodes symétriques.

La relation n:n

On se retrouve avec trois modèles.

Les trois modèles

On pourrait créer un dossier pour les ranger, mais comme il y en a peu, on les gardera comme cela. Si on le faisait, il faudrait adapter les espaces de noms en conséquence. Il faudrait surtout bien renseigner l'espace de noms dans le fichier `config/auth.php` pour le modèle `User`.

La validation

Nous devons considérer un cas de validation un peu particulier. En effet, les mots-clés vont être entrés dans un contrôle de texte séparés par des virgules (on a prévu dans la table `tags` qu'ils ne devraient pas dépasser 50 caractères). Dans l'arsenal des règles de validation de Laravel, on ne dispose pas d'une telle possibilité ; on doit donc la créer.

Ajoutons la règle pour les mots-clés dans la classe `PostRequest` créée au chapitre précédent :

Copyright © 2016 Eyrolles.

```
<?php
namespace App\Http\Requests;

use App\Http\Requests\Request;

class PostRequest extends Request
{
  /**
   * Determine if the user is authorized to make this request.
   *
   * @return bool
   */
  public function authorize()
  {
    return true;
  }

  /**
   * Get the validation rules that apply to the request.
   *
   * @return array
   */
  public function rules()
  {
    return [
      'titre'=>'required|max:80',
      'contenu'=>'required',
      'tags'=>['Regex:/^[A-Za-z0-9-éèàù]{1,50}?(,[A-Za-z0-9-éèàù]
{1,50})*$/']
    ];
  }
}
```

Comme le cas est particulier, j'ai utilisé une expression rationnelle.

Il ne reste plus qu'à traiter le message. Si vous regardez dans le fichier `resources/lang/en/validation.php`, vous trouvez le code suivant :

```
<?php
'custom'=>[
  'attribute-name'=>[
    'rule-name'=>'custom-message',
  ],
],
```

C'est ici qu'on peut ajouter des messages spécifiques. On écrira donc :

```
<?php
'custom'=>[
  'tags'=>[
    'regex'=>"tags, separated by commas (no spaces), should have a maximum
of 50 characters.",
  ],
],
```

Copyright © 2016 Eyrolles.

On fait la même chose dans le fichier du français :

```
<?php
'custom'=>[
  'tags'=>[
    'regex'=>"Les mots-clés, séparés par des virgules (sans espaces),
doivent avoir au maximum 50 caractères alphanumériques.",
  ],
],
```

La gestion

Contrôleur et routes

Vu que tout est en place au niveau des données et de la validation, voyons un peu comment gérer tout cela. On doit compléter le contrôleur `PostController` pour le fonctionnement avec les mots-clés :

```
<?php
namespace App\Http\Controllers;

use App\Repositories\PostRepository;
use App\Repositories\TagRepository;
use App\Http\Requests\PostRequest;

class PostController extends Controller
{
  protected $postRepository;
  protected $nbrPerPage=4;

 public function __construct(PostRepository $postRepository)
  {
    $this->middleware('auth', ['except'=>['index', 'indexTag']]);
    $this->middleware('admin', ['only'=>'destroy']);
    $this->postRepository=$postRepository;
  }

 public function index()
  {
    $posts=$this->postRepository->getWithUserAndTagsPaginate($this-
>nbrPerPage);
    $links=$posts->render();
    return view('posts.liste', compact('posts', 'links'));
  }

 public function create()
  {
    return view('posts.add');
  }
```

Copyright © 2016 Eyrolles.

```
  public function store(PostRequest $request, TagRepository $tagRepository)
  {
    $inputs=array_merge($request->all(), ['user_id'=>$request->user()->id]);
    $post=$this->postRepository->store($inputs);
    if (isset($inputs['tags']))
    {
      $tagRepository->store($post, $inputs['tags']);
    }
    return redirect(route('post.index'));
  }

  public function destroy($id)
  {
    $this->postRepository->destroy($id);
    return redirect()->back();
  }

  public function indexTag($tag)
  {
    $posts=$this->postRepository->getWithUserAndTagsForTagPaginate($tag,
$this->nbrPerPage);
    $links=$posts->render();
    return view('posts.liste', compact('posts', 'links'))
    ->with('info', 'Résultats pour la recherche du mot-clé : ' . $tag);
  }
}
```

J'ai ajouté la méthode `indexTag`. Elle doit lancer la recherche des articles qui comportent le mot-clé passé en paramètre et envoyer les informations dans la vue `liste`. J'ai aussi un peu remanié le code.

Il faut ajouter la route pour aboutir sur cette nouvelle méthode :

```
<?php
Route::resource('post', 'PostController', ['except'=>['show', 'edit',
'update']]);
Route::get('post/tag/{tag}', 'PostController@indexTag');
```

Repositories

Gestionnaire des articles

Voici le repository pour les articles (`app/Lib/Repositories/PostRepository.php`), modifié pour tenir compte des mots-clés :

```
<?php
namespace App\Repositories;

use App\Post;
```

Copyright © 2016 Eyrolles.

```
class PostRepository {
  protected $post;

  public function __construct(Post $post)
  {
    $this->post=$post;
  }

  private function queryWithUserAndTags()
  {
    return $this->post->with('user', 'tags')
    ->orderBy('posts.created_at', 'desc');
  }

  public function getWithUserAndTagsPaginate($n)
  {
    return $this->queryWithUserAndTags()->paginate($n);
  }

  public function getWithUserAndTagsForTagPaginate($tag, $n)
  {
    return $this->queryWithUserAndTags()
    ->whereHas('tags', function($q) use ($tag)
    {
      $q->where('tags.tag_url', $tag);
    })->paginate($n);
  }

  public function store($inputs)
  {
    return $this->post->create($inputs);
  }

  public function destroy($id)
  {
    $post=$this->post->findOrFail($id);
    $post->tags()->detach();
    $post->delete();
  }
}
```

Vous êtes peut-être surpris par la longueur de certains des noms de fonctions. C'est un choix syntaxique. Je préfère des noms explicites, quitte à les allonger.

Gestionnaire des mots-clés

Voici le repository pour les tags (`app/Lib/Repositories/TagRepository.php`) :

```
<?php
namespace App\Repositories;

use App\Tag;
use Illuminate\Support\Str;
```


Copyright © 2016 Eyrolles.

```
class TagRepository
{
  protected $tag;

  public function __construct(Tag $tag)
  {
    $this->tag=$tag;
  }

  public function store($post, $tags)
  {
    $tags=explode(',', $tags);
    foreach ($tags as $tag) {
      $tag=trim($tag);
      $tag_url=Str::slug($tag);
      $tag_ref=$this->tag->where('tag_url', $tag_url)->first();
      if (is_null($tag_ref))
      {
        $tag_ref=new $this->tag([
          'tag'=>$tag,
          'tag_url'=>$tag_url
        ]);
        $post->tags()->save($tag_ref);
      } else
      {
        $post->tags()->attach($tag_ref->id);
      }
    }
  }
}
```

Le fonctionnement

Nous allons à présent analyser ce code.

Liste des articles

La méthode du repository des articles est modifiée et renommée pour ajouter la table `tags` :

```
<?php
public function getWithUserAndTagsPaginate($n)
{
  return $this->queryWithUserAndTags()->paginate($n);
}
```

Pour clarifier le code, j'ai créé une fonction privée qui va nous servir plusieurs fois :

Copyright © 2016 Eyrolles.

```php
<?php
private function queryWithUserAndTags()
{
  return $this->post->with('user', 'tags')
  ->orderBy('posts.created_at', 'desc');
}
```

Vous remarquez qu'on a ajouté la table `tags` comme paramètre de la méthode `with` en plus de `users`. On aura en effet besoin des informations des mots-clés pour l'affichage dans la vue.

Il est intéressant d'étudier les requêtes construites par `Eloquent`, par exemple pour la première page :

```
select count(*) as aggregate from `posts`

select * from `posts` order by `posts`.`created_at` desc limit 4 offset 0

select * from `users` where `users`.`id` in ('7','2','6')

select `tags`.*, `post_tag`.`post_id` as `pivot_post_id`, `post_
tag`.`tag_id` as `pivot_tag_id` from `tags` inner join `post_
tag` on `tags`.`id`=`post_tag`.`tag_id` where `post_tag`.`post_id`
in ('25','1','18','14')
```

On demande :

- le nombre total d'articles pour la pagination ;
- les quatre premières lignes des articles avec l'ordre des dates ;
- les utilisateurs qui correspondent aux articles sélectionnés ;
- les tags concernés par les articles.

On se rend compte là du travail effectué par `Eloquent` pour nous !

Nouvel article

L'enregistrement d'un nouvel article sera évidemment un peu plus délicat à cause de la présence des mots-clés. Dans le repository des articles, on se contente d'enregistrer le nouvel élément :

```php
<?php
public function store($inputs)
{
  return $this->post->create($inputs);
}
```

C'est dans le repository des mots-clés qu'a lieu le plus gros du travail :

Copyright © 2016 Eyrolles.

```
<?php
public function store($post, $tags)
{
  $tags=explode(',', $tags);
  foreach ($tags as $tag) {
    $tag=trim($tag);
    $tag_url=Str::slug($tag);
    $tag_ref=$this->tag->where('tag_url', $tag_url)->first();
    if (is_null($tag_ref))
    {
      $tag_ref=new $this->tag([
        'tag'=>$tag,
        'tag_url'=>$tag_url
      ]);
      $post->tags()->save($tag_ref);
    } else
    {
      $post->tags()->attach($tag_ref->id);
    }
  }
}
```

Ce code mérite quelques commentaires. Les mots-clés sont envoyés par le formulaire (que nous verrons plus loin) sous la forme de texte avec comme séparateur une virgule, par exemple : `tag1, tag2, tag3`.

Dans le contrôleur, la première chose est de vérifier que des mots-clés ont bien été saisis :

```
<?php
if (isset($inputs['tags']))
{
  $tagRepository->store($post, $inputs['tags']);
}
```

Si c'est le cas, on appelle la méthode `store` du repository en transmettant les mots-clés et une référence du modèle créé. Dans le repository on crée un tableau en utilisant la virgule comme séparateur :

```
<?php
$tags=explode(',', $tags);
```

Ensuite, on parcourt le tableau :

```
<?php
foreach ($tags as $tag)
```

Par précaution, on supprime les espaces éventuels :

```
<?php
$tag=trim($tag);
```

Copyright © 2016 Eyrolles.

On crée la version pour URL du mot-clé (avec la méthode `slug` de la classe `Str`) :

```
<?php
$tag_url=Str::slug($tag);
```

On vérifie si ce mot-clé existe déjà :

```
<?php
$tag_ref=$this->tag->where('tag_url', $tag_url)->first();
```

Si ce n'est pas le cas, on le crée :

```
<?php
$tag_ref=new $this->tag([
  'tag'=>$tag,
  'tag_url'=>$tag_url
]);
$post->tags()->save($tag_ref);
```

Remarquez comment la méthode `save` ici permet à la fois de créer le mot-clé et de référencer la table pivot.

Si le mot-clé existe déjà, on se contente d'informer la table pivot avec la méthode `attach` :

```
<?php
$post->tags()->attach($tag_ref->id);
```

Suppression d'un article

Quand on supprime un article, il faut aussi supprimer ses liens avec les mots-clés :

```
<?php
public function destroy($id)
{
  $post=$this->post->findOrFail($id);
  $post->tags()->detach();
  $post->delete();
}
```

La méthode `detach` supprime les lignes dans la table pivot.

Recherche par mot-clé

Il nous reste enfin à décrire la recherche par sélection d'un mot-clé :

Copyright © 2016 Eyrolles.

```
<?php
public function getWithUserAndTagsForTagPaginate($tag, $n)
{
  return $this->queryWithUserAndTags()
  ->whereHas('tags', function($q) use ($tag)
  {
    $q->where('tags.tag_url', $tag);
  })->paginate($n);
}
```

Vous remarquez que, par rapport au code de la méthode `getWithUserAndTagsPaginate`, on a ajouté la méthode `whereHas`. Cette dernière permet d'ajouter une condition sur une table chargée. Il est intéressant là aussi d'analyser les requêtes construites par `Eloquent` :

```
select count(*) as aggregate from `posts` where (select count(*) from `tags`
inner join `post_tag` on `tags`.`id`=`post_tag`.`tag_id` where `post_
tag`.`post_id`=`posts`.`id` and `tags`.`tag_url`='tag-14') >= '1'

select * from `posts` where (select count(*) from `tags` inner
join `post_tag` on `tags`.`id`=`post_tag`.`tag_id` where `post_
tag`.`post_id`=`posts`.`id` and `tags`.`tag_url`='tag-14') >= '1'
order by `posts`.`created_at` desc limit 4 offset 0

select * from `users` where `users`.`id` in ('3','1','6','4')
select `tags`.*, `post_tag`.`post_id` as `pivot_post_id`, `post_
tag`.`tag_id` as `pivot_tag_id` from `tags` inner join `post_
tag` on `tags`.`id`=`post_tag`.`tag_id` where `post_tag`.`post_id`
in ('13','76','87','37')
```

Les cinq requêtes sont plutôt chargées :

- on compte les enregistrements pour la pagination (avec une jointure) ;
- on récupère les 4 lignes des articles (avec une jointure) ;
- on récupère les utilisateurs rédacteurs des articles ;
- on récupère les mots-clés concernés par les articles (avec une jointure).

Il y a une chose que je n'ai pas prise en compte dans tout ce code : c'est le cas des mots-clés orphelins, en cas de suppression d'un article. Cette gestion n'est pas obligatoire ; on peut prévoir une maintenance épisodique de la base ou une action de l'administrateur.

Les vues

Template

On conserve le même template (`resources/views/template.blade.php`) :

Copyright © 2016 Eyrolles.

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Mon joli site</title>
    {!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/
bootstrap.min.css') !!}
    {!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/
bootstrap-theme.min.css') !!}
    <!--[if lt IE 9]>
      {{ Html::style('https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.
js') }}
      {{ Html::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.
min.js') }}
    <![endif]-->
    <style>textarea {resize:none;}</style>
  </head>
  <body>
    <header class="jumbotron">
      <div class="container">
        <h1 class="page-header">{!! link_to_route('post.index', 'Mon joli
blog') !!}</h1>
        @yield('header')
      </div>
    </header>
    <div class="container">
      @yield('contenu')
    </div>
  </body>
</html>
```

Liste

Voici la vue pour la liste des articles (`resources/views/posts/liste.blade.php`) :

```
@extends('template')

@section('header')
  @if(Auth::check())
    <div class="btn-group pull-right">
      {!! link_to_route('post.create', 'Créer un article', [],
['class'=>'btn btn-info']) !!}
      {!! link_to('logout', 'Deconnexion', ['class'=>'btn btn-warning']) !!}
    </div>
  @else
    {!! link_to('login', 'Se connecter', ['class'=>'btn btn-info pull-
right']) !!}
  @endif
@endsection
```

Copyright © 2016 Eyrolles.

```
@section('contenu')
  @if(isset($info))
    <div class="row alert alert-info">{{$info}}</div>
  @endif
  {!! $links !!}
  @foreach($posts as $post)
    <article class="row bg-primary">
      <div class="col-md-12">
        <header>
          <h1>{{$post->titre}}
            <div class="pull-right">
              @foreach($post->tags as $tag)
                {!! link_to('post/tag/' . $tag->tag_url, $tag->tag,
['class'=>'btn btn-xs btn-info']) !!}
              @endforeach
            </div>
          </h1>
        </header>
        <hr>
        <section>
          <p>{{$post->contenu}}</p>
          @if(Auth::check() and Auth::user()->admin)
            {!! Form::open(['method'=>'DELETE', 'route'=>['post.destroy',
$post->id]]) !!}
              {!! Form::submit('Supprimer cet article', ['class'=>'btn
btn-danger btn-xs ', 'onclick'=>'return confirm(\'Vraiment supprimer cet
article ?\')']) !!}
            {!! Form::close() !!}
          @endif
          <em class="pull-right">
            <span class="glyphicon glyphicon-pencil"></span>{{$post->user-
>name}} le {!! $post->created_at->format('d-m-Y') !!}
          </em>
        </section>
      </div>
    </article>
    <br>
  @endforeach
  {!!$links!!}
@endsection
```

Copyright © 2016 Eyrolles.

La liste des articles

Les mots-clés apparaissent sous forme de petits boutons. L'action de cliquer sur un de ces boutons lance la recherche à partir de ce mot-clé et affiche les articles correspondants, ainsi qu'une barre d'information.

La recherche par mot-clé

Un utilisateur connecté dispose en plus du bouton pour créer un article. L'administrateur a en plus le bouton de suppression.

Copyright © 2016 Eyrolles.

Les boutons pour les utilisateurs connectés

Vue de création d'un article

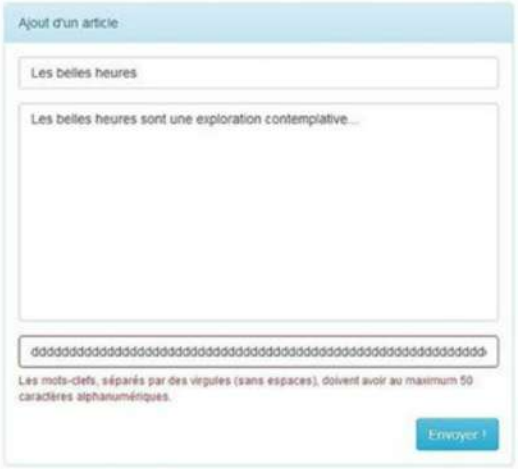
Le formulaire de création d'un article (`resources/views/posts/add.blade.php`) a été enrichi d'un contrôle de texte pour la saisie des mots-clés :

```
@extends('template')

@section('contenu')
  <br>
  <div class="col-sm-offset-3 col-sm-6">
    <div class="panel panel-info">
      <div class="panel-heading">Ajout d'un article</div>
      <div class="panel-body">
        {!! Form::open(['route'=>'post.store']) !!}
          <div class="form-group {!! $errors->has('titre') ? 'has-
error' : '' !!}">
            {!! Form::text('titre', null, ['class'=>'form-control',
'placeholder'=>'Titre']) !!}
            {!! $errors->first('titre', '<small class="help-block">:message</
small>') !!}
          </div>
          <div class="form-group {!! $errors->has('contenu') ? 'has-
error' : '' !!}">
            {!! Form::textarea ('contenu', null, ['class'=>'form-control',
'placeholder'=>'Contenu']) !!}
            {!! $errors->first('contenu', '<small class="help-
block">:message</small>') !!}
          </div>
          <div class="form-group {{ $errors->has('tags') ? 'has-
error' : '' }}">
            {!! Form::text('tags', null, array('class'=>'form-control',
'placeholder'=>'Entrez les mots-clés séparés par des virgules')) !!}
```

Copyright © 2016 Eyrolles.

```
            {!! $errors->first('tags', '<small class="help-block">:message</
small>') !!}
          </div>
          {!! Form::submit('Envoyer !', ['class'=>'btn btn-info pull-
right']) !!}
        {!! Form::close() !!}
      </div>
    </div>
    <a href="javascript:history.back()" class="btn btn-primary">
      <span class="glyphicon glyphicon-circle-arrow-left"></span>Retour
    </a>
  </div>
@endsection
```

Le message d'erreur pour la validation des mots-clés est aussi géré.



Le formulaire de création d'un article

Je ne détaille pas le code de toutes ces vues ; il n'est pas bien compliqué et recouvre des situations déjà rencontrées.

En résumé

- Une relation de type n:n nécessite la création d'une table pivot.
- `Eloquent` gère élégamment les tables pivots avec des méthodes adaptées.
- On peut créer des règles et des messages de validation personnalisés.

Copyright © 2016 Eyrolles.

16 Les commandes et les assistants

Dans les chapitres précédents, on a dû à plusieurs reprises créer des migrations, des populations, des contrôleurs, des modèles… `Artisan` possède des commandes pour effectuer certaines de ces opérations, mais il ne va pas bien loin et il nous a fallu créer beaucoup de code qui, de toute évidence, pourrait plus ou moins facilement être automatisé.

Il est possible d'améliorer les commandes d'`Artisan` ou même de s'en créer des nouvelles. Il existe aussi des assistants pour nous aider dans ces tâches un peu pénibles ou répétitives.

Améliorer une commande

Il existe dans `Artisan` une commande pour créer un modèle :

```
php artisan make:model MonModele
```

Voici ce qu'on obtient :

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class MonModele extends Model
{
  //
}
```

Copyright © 2016 Eyrolles.

Ce n'est qu'une coquille vide, mais c'est déjà ça.

Cette commande permet toutefois de lancer une migration avec l'option `--migration`.

Si on a prévu de placer les modèles dans un dossier spécifique, il suffit de renseigner l'espace de noms dans la commande :

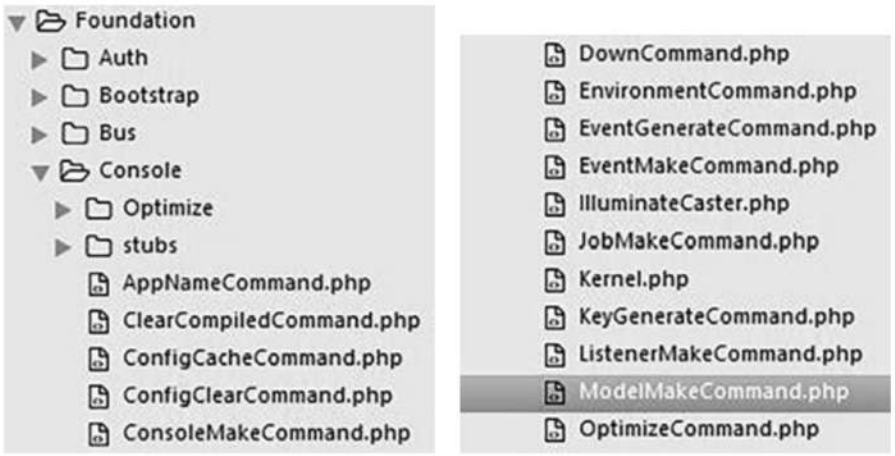
```
php artisan make:model App\Models\MonModele
```

On a vu qu'il était pratique d'avoir une propriété `$fillable` pour répertorier les colonnes qu'on peut mettre à jour sans risque avec une affectation de masse. Dans le dossier des commandes console, il existe déjà une option qui permet d'ajouter cette propriété.

Les commandes console

Il s'agit d'un exemple qui peut vous guider pour réaliser une commande.

Puisque la commande existe déjà dans `Artisan`, on ne la réinventera pas, mais nous nous contenterons d'étendre ses possibilités. Il faut un peu fouiller dans le framework pour trouver cette commande.



La commande ModelMakeCommand de Laravel

Voici son code :

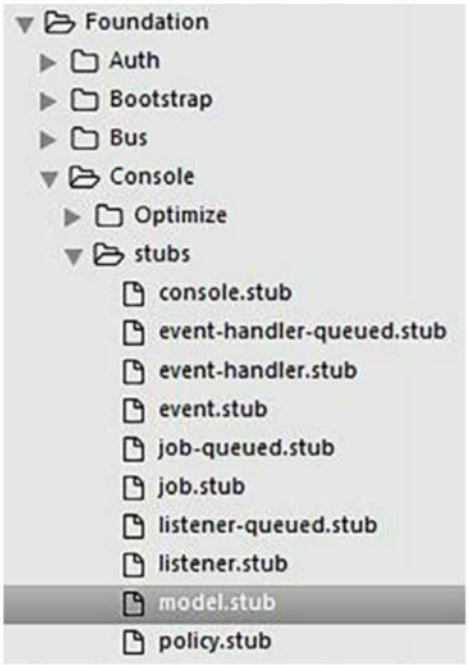
Copyright © 2016 Eyrolles.

```
<?php
namespace Illuminate\Foundation\Console;

use Illuminate\Console\GeneratorCommand;
use Symfony\Component\Console\Input\InputOption;

class ModelMakeCommand extends GeneratorCommand
{
  /**
   * The console command name.
   *
   * @var string
   */
  protected $name='make:model';

  /**
   * The console command description.
   *
   * @var string
   */
  protected $description='Create a new Eloquent model class';

  /**
   * The type of class being generated.
   *
   * @var string
   */
  protected $type='Model';

  /**
   * Execute the console command.
   *
   * @return void
   */
  public function fire()
  {
    if (parent::fire()!==false) {
      if ($this->option('migration')) {
        $table=str_plural(snake_case(class_basename($this-
>argument('name'))));
        $this->call('make:migration', ['name'=>"create_{$table}_table",
'--create'=>$table]);
      }
    }
  }
 /**
   * Get the stub file for the generator.
   *
   * @return string
   */
  protected function getStub()
  {
    return __DIR__.'/stubs/model.stub';
  }
```

Copyright © 2016 Eyrolles.

```
    /**
     * Get the default namespace for the class.
     *
     * @param  string  $rootNamespace
     * @return string
     */
    protected function getDefaultNamespace($rootNamespace)
    {
        return $rootNamespace;
    }

    /**
     * Get the console command options.
     *
     * @return array
     */
    protected function getOptions()
    {
        return [
            ['migration', 'm', InputOption::VALUE_NONE, 'Create a new migration
file for the model.'],
        ];
    }
}
```

La commande utilise un gabarit (`stub`) qu'on trouve à l'emplacement suivant.



Le stub de la commande

Copyright © 2016 Eyrolles.

En voici le code :

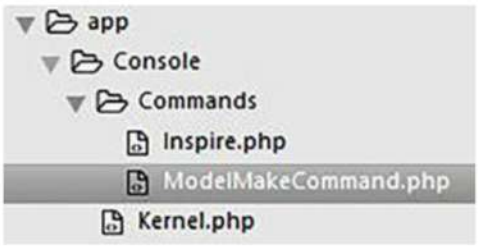
```
<?php
namespace DummyNamespace;

use Illuminate\Database\Eloquent\Model;

class DummyClass extends Model
{
  //
}
```

Les `Dummy` permettent d'avoir des emplacements aux données variables, ici le nom de l'espace de noms et celui de la classe.

Dans la version 5.0, on avait une autre syntaxe constituée d'accolades comme pour `Blade`.

On va commencer par créer notre classe en étendant celle de Laravel. Alors, créons la nouvelle commande avec `Artisan` :

```
php artisan make:console ModelMakeCommand
```



Notre commande

Voici le code généré :

```
<?php
namespace App\Console\Commands;

use Illuminate\Console\Command;

class ModelMakeCommand extends Command
{
 /**
```

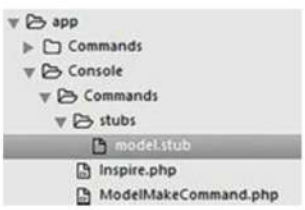
Copyright © 2016 Eyrolles.

```
    * The name and signature of the console command.
    *
    * @var string
    */
   protected $signature='command:name';

   /**
    * The console command description.
    *
    * @var string
    */
   protected $description='Command description.';

   /**
    * Create a new command instance.
    *
    * @return void
    */
   public function __construct()
   {
     parent::__construct();
   }

   /**
    * Execute the console command.
    *
    * @return mixed
    */
   public function handle()
   {
     //
   }
}
```

On l'enregistre dans `app\Console\Kernel.php` :

```
<?php
protected $commands=[
  Commands\Inspire::class,
  Commands\ModelMakeCommand::class,
];
```

On créera aussi notre `stub`.



Notre stub

Copyright © 2016 Eyrolles.

Voici le code pour ajouter la propriété `$fillable` :

```
<?php
namespace DummyNamespace;

use Illuminate\Database\Eloquent\Model;

class DummyClass extends Model {
  /**
   * Attributes that should be mass-assignable.
   *
   * @var array
   */
  protected $fillable=[DummyFillable];
}
```

Il ne nous reste plus qu'à modifier le code généré automatiquement pour, d'une part, étendre la classe de Laravel et d'autre part, assurer le fonctionnement de l'option `fillable` :

```
<?php
namespace App\Console\Commands;

use Illuminate\Foundation\Console\ModelMakeCommand as BaseModelCommand;
use Symfony\Component\Console\Input\InputOption;
use Illuminate\Support\Facades\Schema as Schema;

class ModelMakeCommand extends BaseModelCommand
{
 protected $exclude=['id', 'password', 'created_at', 'updated_at'];

  protected function getStub()
  {
    return __DIR__.'/stubs/model.stub';
  }

 protected function buildClass($name)
  {
    $stub=$this->files->get($this->getStub());
    return $this->replaceNamespace($stub, $name)
    ->replaceFillable($stub)
    ->replaceClass($stub, $name);
  }

 protected function replaceFillable(&$stub)
  {
    if ($this->input->getOption('fillable'))
    {

    // On construit le nom de la table à partir de celui du modèle.
      $table=str_plural(strtolower($this->getNameInput()));
     // On récupère le nom des colonnes.
      $columns=Schema::getColumnListing($table);
    // On exclut les colonnes non désirées.
    $columns=array_filter($columns, function($value)
```

Copyright © 2016 Eyrolles.

```
        {
          return !in_array($value, $this->exclude);
        });
        // On ajoute des apostrophes.
        array_walk($columns, function(&$value) {
          $value="'" . $value . "'";
        });
        // CSV format
        $columns=implode(',', $columns);
      }
      $stub=str_replace('DummyFillable', isset($columns)? $columns : '',
$stub);
      return $this;
    }

    protected function getOptions()
    {
      return [
        ['migration', 'm', InputOption::VALUE_NONE, 'Create a new migration
file for the model.'],
        ['fillable', null, InputOption::VALUE_NONE, 'Set the fillable columns.',
null]
      ];
    }
}
```

Je ne détaillerai pas le code de cette commande ; je vous laisse l'analyser pour en comprendre le fonctionnement. Le but est juste de montrer qu'il est possible de le réaliser avec un exemple concret.

Testons cette commande avec la table `posts` qui nous a servi pour le petit blog :

```
php artisan make:model Post --fillable
```

On obtient le modèle suivant :

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model {
    /**
     *Attributes that should be mass-assignable.
     *
     * @var array
     */
    protected $fillable=['titre','contenu','user_id'];
}
```

On a bien créé la propriété et le remplissage du tableau avec exclusion des colonnes prévues.

Copyright © 2016 Eyrolles.

Laravel Schema Designer

Le designer de schéma (http://www.laravelsd.com/) est une démarche visuelle et efficace. Il suffit de dessiner les tables et de renseigner champs et relations ; ensuite sont créés pour nous migrations, populations et modèles.

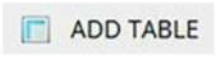
Prenons un exemple avec des livres et des auteurs, liés par une relation 1:n.

Tout d'abord, il faut créer un compte, ce qui est gratuit. Ensuite, vous donnez un nom à votre base.

Le nom de la base

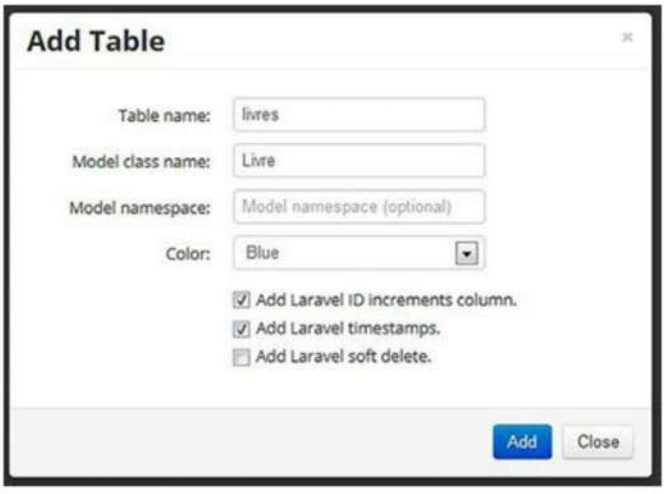
Création des tables

Cliquez sur le bouton d'ajout de table.



Créer une table

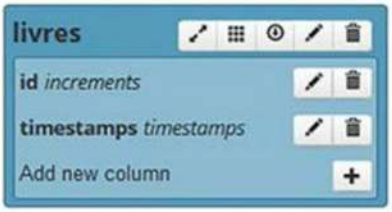
Renseignez les champs.



Le formulaire de création d'une table

Vous obtenez alors votre table visuellement.

Copyright © 2016 Eyrolles.



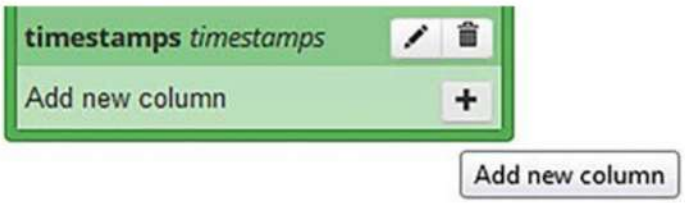
La table des livres

Créez de la même manière la table des auteurs (vous pouvez jouer avec les couleurs pour bien distinguer les tables, ce qui s'avère utile dès qu'il y en a beaucoup).

La table des auteurs

Création des champs

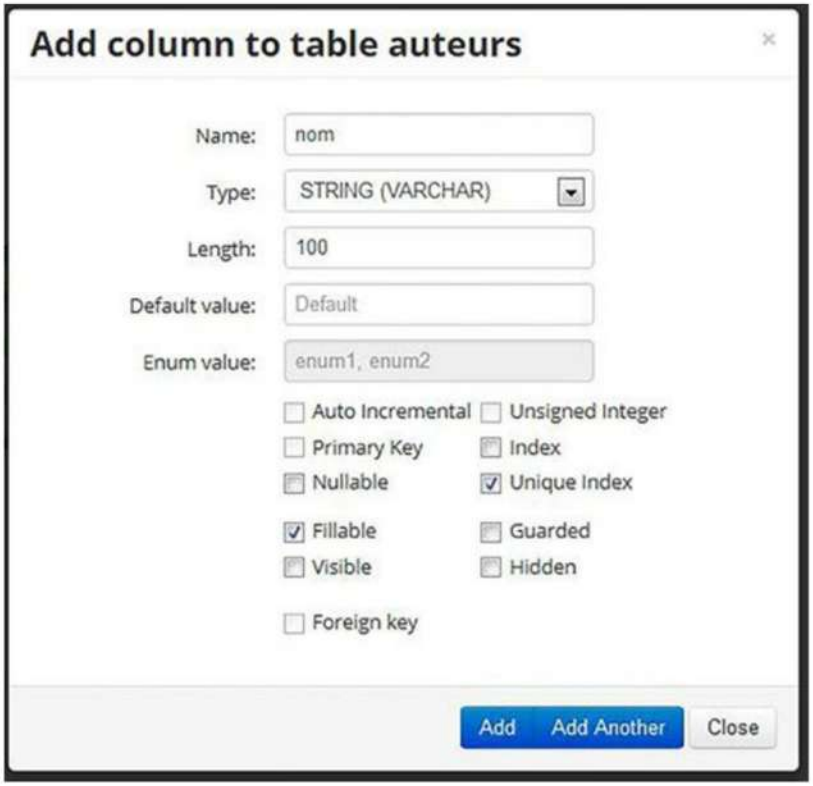
On renseigne ensuite les champs pour les tables. Commençons par les auteurs. Il suffit de cliquer sur le bouton comportant le signe +.



Ajout d'un champ

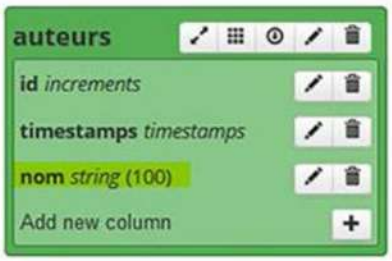
Le formulaire est très complet, il suffit de le renseigner.

Copyright © 2016 Eyrolles.



Le formulaire de création d'un champ

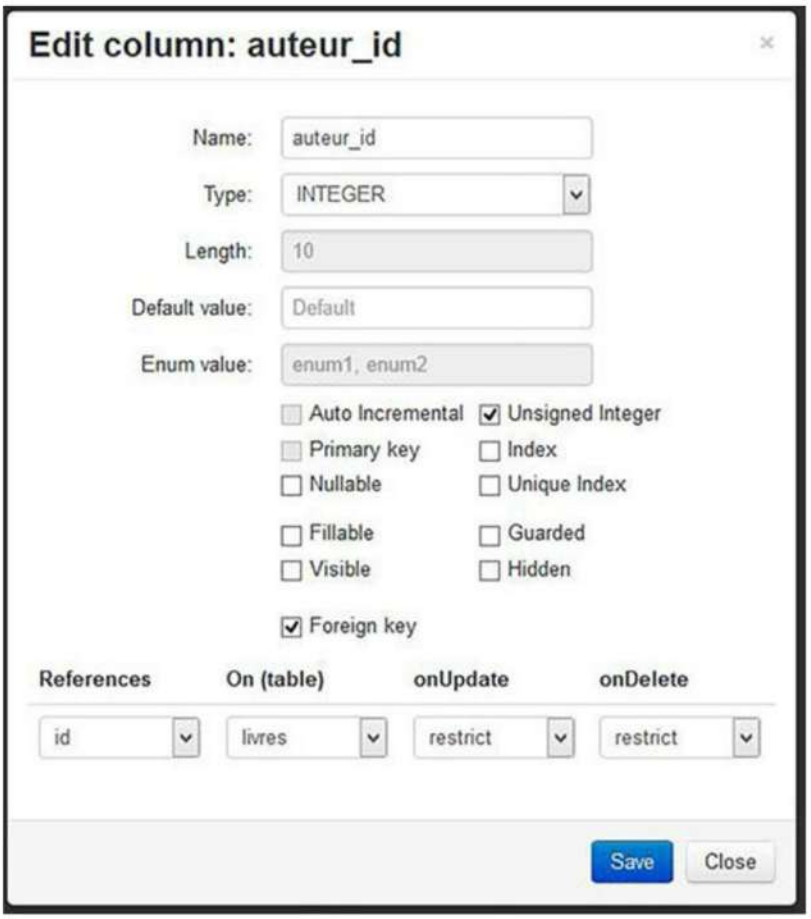
Après validation, on voit notre nouveau champ dans la table.



Le champ ajouté

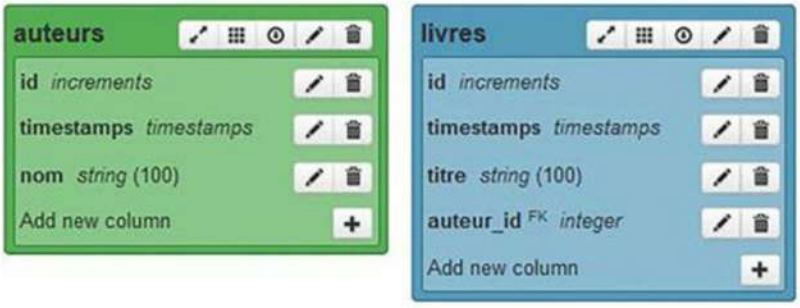
Deux boutons sont présents pour le modifier ou le supprimer. De la même manière, on renseigne la table des livres avec le titre et la clé étrangère.

Copyright © 2016 Eyrolles.



La clé étrangère

On se retrouve donc avec nos deux tables et leurs champs renseignés.

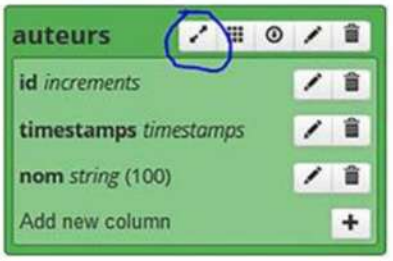


Les deux tables créées avec leurs champs renseignés

Création des relations

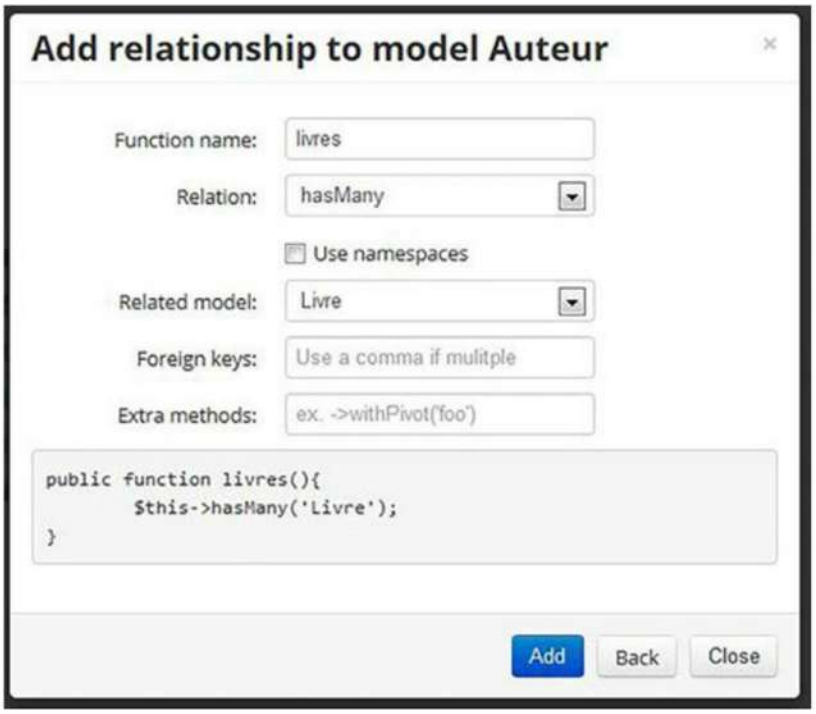
Il ne nous reste plus qu'à définir les relations. Commençons par les auteurs. Il faut cliquer sur le bouton suivant.

Copyright © 2016 Eyrolles.



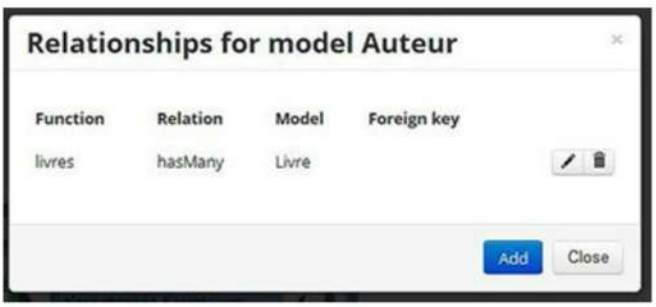
Le bouton de création d'une relation

On obtient alors un formulaire à compléter.



Le formulaire de création de la relation

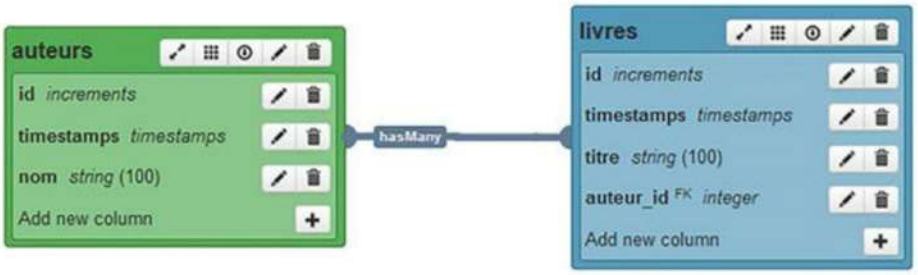
Ensuite, la liste des relations existantes est affichée.



Les relations existantes

Copyright © 2016 Eyrolles.

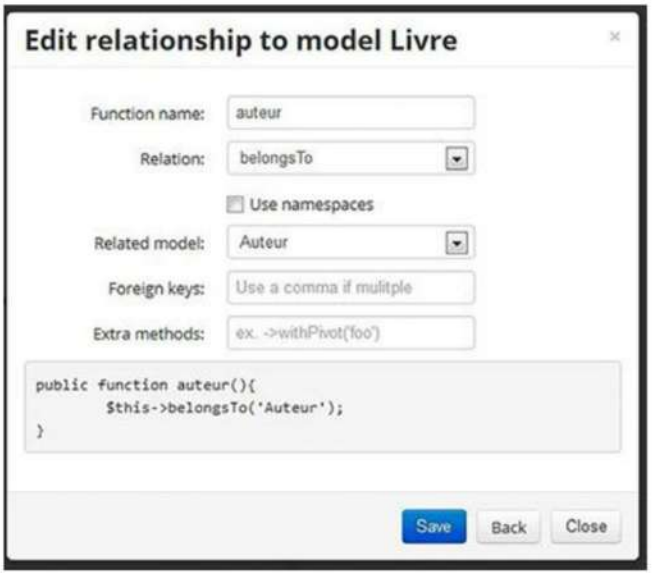
Évidemment, nous n'en avons qu'une. La relation est représentée par une liaison avec apparition du nom de la méthode au survol de la souris.



La relation représentée visuellement

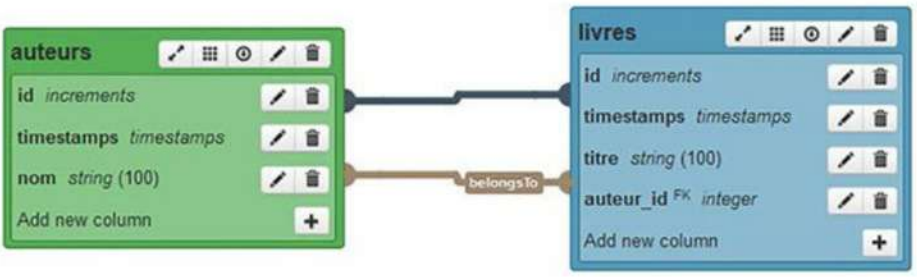
Voilà qui est bien pratique, n'est-ce pas ?!

De la même manière, on crée la relation vue du côté des livres.



La relation vue du côté des livres

On a également une représentation visuelle avec le nom de la méthode qui apparaît au survol.



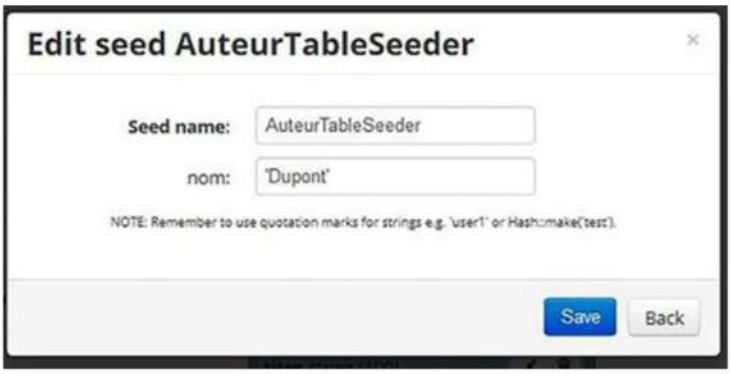
La représentation visuelle de la méthode belongsTo

Ajoutons la population.

Copyright © 2016 Eyrolles.

Le bouton pour la population

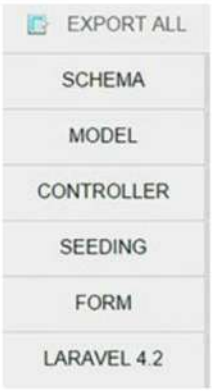
Voici le formulaire à compléter.



Le formulaire pour la population

Exportation des fichiers

Notre schéma est terminé. Il ne reste plus qu'à exporter les fichiers.



Le menu de l'exportation

En cliquant directement sur ***EXPORT ALL***, on reçoit un fichier compressé contenant tous les fichiers.

Copyright © 2016 Eyrolles.

Les fichiers exportés

Voici la migration pour les auteurs :

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;

class CreateAuteursTable extends Migration {
  public function up()
  {
    Schema::create('auteurs', function(Blueprint $table) {
      $table->increments('id');
      $table->timestamps();
      $table->string('nom', 100)->unique();
    });
  }

  public function down()
  {
    Schema::drop('auteurs');
  }
}
```

Une migration spécifique des clés étrangères :

Copyright © 2016 Eyrolles.

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;

class CreateForeignKeys extends Migration {
  public function up()
  {
    Schema::table('livres', function(Blueprint $table) {
      $table->foreign('auteur_id')->references('id')->on('livres')
            ->onDelete('restrict')
            ->onUpdate('restrict');
    });
  }

  public function down()
  {
    Schema::table('livres', function(Blueprint $table) {
      $table->dropForeign('livres_auteur_id_foreign');
    });
  }
}
```

Voici la population qu'on a prévue pour un auteur :

```
<?php
class AuteurTableSeeder extends Seeder {
  public function run()
  {
    // DB::table('auteurs')->delete();

    // AuteurTableSeeder
    Auteur::create(array(
      'nom'=>'Dupont'
    ));
  }
}
```

Le `DatabaseSeeder` est à jour :

```
<?php
use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class DatabaseSeeder extends Seeder {
  public function run()
  {
    Model::unguard();
    $this->call('AuteurTableSeeder');
    $this->command->info('Auteur table seeded!');
  }
}
```

Copyright © 2016 Eyrolles.

Voici le modèle des auteurs :

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Auteur extends Model {
  protected $table='auteurs';
  public $timestamps=true;
  protected $fillable=array('nom');

  public function livres()
  {
    return $this->hasMany('Livre');
  }
}
```

On a même des vues avec un squelette de formulaire, par exemple pour les auteurs :

```
{!! Form::open(array('route'=>'route.name', 'method'=>'POST')) !!}
  <ul>
    <li>
      {!! Form::label('nom', 'Nom:') !!}
      {!! Form::text('nom') !!}
    </li>
    <li>
      {!! Form::submit() !!}
    </li>
  </ul>
{!! Form::close() !!}
```

Ce générateur crée aussi des contrôleurs et est vraiment très convivial avec son interface graphique.

En résumé

- `Artisan` propose un grand arsenal de commandes pour générer du code. Il est possible de créer de nouvelles commandes ou d'étendre les possibilités des commandes existantes.
- À partir d'une interface visuelle conviviale, le designer de schéma facilite la création des migrations, des modèles, des populations, des contrôleurs et des vues.

Copyright © 2016 Eyrolles.

17 Query Builder

Nous avons jusqu'à présent utilisé `Eloquent` pour générer nos requêtes. Malgré sa pertinence, il est parfois nécessaire de créer des requêtes qui dépassent ses compétences. C'est alors qu'intervient le Query Builder, un outil pratique avec une syntaxe explicite, qui est le parfait compagnon d'`Eloquent`.

Les données

Migration

Pour effectuer des tests, nous aurons besoin de données. Créez une nouvelle migration :

```
php artisan make:migration create_livres_table
```

Et entrez ce code :

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateLivresTable extends Migration {
  /**
   * Run the migrations.
   *
   * @return void
   */
  public function up()
 {
    Schema::create('auteurs', function(Blueprint $table) {
      $table->increments('id');
```

Copyright © 2016 Eyrolles.

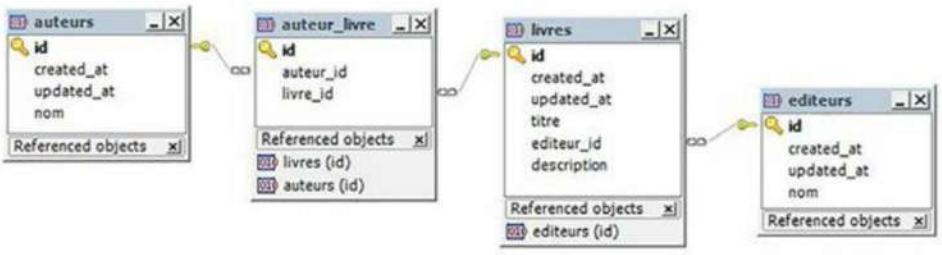
```
            $table->timestamps();
            $table->string('nom', 100)->unique();
        });
        Schema::create('editeurs', function(Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
            $table->string('nom', 100)->unique();
        });
        Schema::create('livres', function(Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
            $table->string('titre', 100);
            $table->integer('editeur_id')->unsigned();
            $table->text('description');
        });
        Schema::create('auteur_livre', function(Blueprint $table) {
            $table->increments('id');
            $table->integer('auteur_id')->unsigned();
            $table->integer('livre_id')->unsigned();
        });
        Schema::table('livres', function(Blueprint $table) {
            $table->foreign('editeur_id')->references('id')->on('editeurs')
              ->onDelete('restrict')
              ->onUpdate('restrict');
        });
        Schema::table('auteur_livre', function(Blueprint $table) {
            $table->foreign('auteur_id')->references('id')->on('auteurs')
              ->onDelete('restrict')
              ->onUpdate('restrict');
        });
        Schema::table('auteur_livre', function(Blueprint $table) {
            $table->foreign('livre_id')->references('id')->on('livres')
              ->onDelete('restrict')
              ->onUpdate('restrict');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('livres', function(Blueprint $table) {
            $table->dropForeign('livres_editeur_id_foreign');
        });
        Schema::table('auteur_livre', function(Blueprint $table) {
            $table->dropForeign('auteur_livre_auteur_id_foreign');
        });
        Schema::table('auteur_livre', function(Blueprint $table) {
            $table->dropForeign('auteur_livre_livre_id_foreign');
        });
        Schema::drop('auteur_livre');
        Schema::drop('livres');
        Schema::drop('auteurs');
        Schema::drop('editeurs');
    }
}
```

Copyright © 2016 Eyrolles.

Lancez la migration :

```
php artisan migrate
```

Le résultat est la création des quatre tables suivantes avec leurs relations : 1:n entre éditeurs et livres, n:n entre auteurs et livres.



Les quatre tables et leurs relations

Population

Pour la population, nous allons utiliser la bibliothèque *fzaninotto/Faker* (*https://github.com/fzaninotto/Faker*), qui est chargée par défaut par Laravel. Elle facilite la génération des noms, adresses, textes, nombres… C'est très pratique par exemple lorsqu'on effectue des tests.

Laravel intègre cette bibliothèque et permet de créer des *Model Factories*.

Le ModelFactory

Voici un exemple pour le modèle d'utilisateur :

```
<?php
/* |--------------------------------------------------------------------------
   | Model Factories
   |--------------------------------------------------------------------------
   |
   | Here you may define all of your model factories. Model factories give
   | you a convenient way to create models for testing and seeding your
   | database. Just tell the factory how a default model should look.
   |
 */
```

Copyright © 2016 Eyrolles.

```
$factory->define(App\User::class, function (Faker\Generator $faker) {
  return [
    'name'=>$faker->name,
    'email'=>$faker->email,
    'password'=>bcrypt(str_random(10)),
    'remember_token'=>str_random(10),
  ];
});
```

Créons nos *factories* pour nos modèles :

```
<?php
$factory->define(App\Editeur::class, function (Faker\Generator $faker) {
  return [
    'nom'=>$faker->name,
  ];
});

$factory->define(App\Auteur::class, function (Faker\Generator $faker) {
  return [
    'nom'=>$faker->name,
  ];
});

$factory->define(App\Livre::class, function (Faker\Generator $faker) {
  return [
    'titre'=>$faker->sentence(3),
    'description'=>$faker->text,
    'editeur_id'=>$faker->numberBetween(1,40),
  ];
});
```

Ensuite, on prévoit le code suivant dans `DatabaseSeeder` :

```
<?php
use Illuminate\Database\Seeder;
use Faker\Factory;

class DatabaseSeeder extends Seeder {
  /**
   * Run the database seeds.
   *
   * @return void
   */
  public function run()
  {
    factory(App\Editeur::class, 40)->create();
    factory(App\Auteur::class, 40)->create();
    factory(App\Livre::class, 80)->create();
    for ($i=1; $i<41; $i++) {
      $number=rand(2,8);
```

Copyright © 2016 Eyrolles.

```
        for ($j=1; $j<=$number; $j++) {
          DB::table('auteur_livre')->insert([
            'livre_id'=>rand(1,40),
            'auteur_id'=>$i
          ]);
        }
      }
    }
}
```

Lancez ensuite la population :

```
php artisan db:seed
```

On obient ainsi 40 éditeurs et 40 auteurs avec des noms aléatoires, ainsi que 80 livres liés aux éditeurs et aux auteurs.

Il nous faut aussi les modèles pour faire marcher tout cela si on veut utiliser `Eloquent`.

Modèle pour les éditeurs

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Editeur extends Model
{
  public function livres()
  {
    return $this->hasMany('App\Livre');
  }
}
```

Modèle pour les auteurs

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Auteur extends Model
{
  public function livres()
  {
    return $this->belongsToMany('App\Livre');
  }
}
```

Copyright © 2016 Eyrolles.

Modèle pour les livres

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Livre extends Model
{
  public function auteurs()
  {
    return $this->belongsToMany('App\Auteur');
  }

  public function editeur()
  {
    return $this->belongsTo('App\Editeur');
  }
}
```

Les sélections

Liste de tous les éditeurs

Avec Eloquent

```
<?php
$editeurs=App\Editeur::all();
foreach ($editeurs as $editeur) {
  echo $editeur->nom, '<br>';
}
```

Avec le Query Builder

```
<?php
$editeurs=DB::table('editeurs')->get();
foreach ($editeurs as $editeur) {
    echo $editeur->nom, '<br>';
}
```

Il faut bien désigner la table et utiliser ensuite la méthode `get`. Le résultat est le même.

Copyright © 2016 Eyrolles.

Tableau des valeurs d'une colonne

Avec Eloquent

```
<?php
$editeur=App\Editeur::lists('nom');
foreach ($editeurs as $editeur) {
    echo $editeur, '<br>';
}
```

Avec le Query Builder

```
<?php
$editeurs=DB::table('editeurs')->lists('nom');
foreach ($editeurs as $editeur) {
  echo $editeur, '<br>';
}
```

Requête SQL correspondante

```
select nom from editeurs
```

Ligne particulière

Cherchons par exemple l'éditeur dont l'`id` est 10.

Avec Eloquent

```
<?php
$editeur=App\Editeur::find(10);
```

Avec le Query Builder

```
<?php
$editeur=DB::table('editeurs')->whereId(10)->first();
echo $editeur->nom;
```

On précise la table, puis l'`id` avec la méthode `where` et on prend le premier enregistrement (`first`).

Requête SQL correspondante

```
SELECT * FROM editeurs WHERE id=10
```

Copyright © 2016 Eyrolles.

Colonne isolée

Avec le Query Builder

```
<?php
$editeur=DB::table('editeurs')->whereId(10)->pluck('nom');
dd($editeur);
```

Requête SQL correspondante

```
select nom from editeurs where id=10
```

On peut aussi sélectionner des colonnes avec un `select` :

Avec Eloquent

```
<?php
$editeurs=App\Editeur::select('nom')->get();
foreach ($editeurs as $editeur) {
  echo $editeur->nom, '<br>';
}
```

Avec le Query Builder

```
<?php
$editeurs=DB::table('editeurs')->select('nom')->get();
foreach ($editeurs as $editeur) {
  echo $editeur->nom, '<br>';
}
```

Requête SQL correspondante

```
select nom from editeurs
```

Lignes distinctes

Avec Eloquent

```
<?php
$livres=App\Livre::select('editeur_id')->distinct()->get();
foreach ($livres as $livre) {
  echo $livre->editeur_id, '<br>';
}
```

Copyright © 2016 Eyrolles.

Avec le Query Builder

```
<?php
$livres=DB::table('livres')->select('editeur_id')->distinct()->get();
foreach ($livres as $livre) {
  echo $livre->editeur_id, '<br>';
}
```

Requête SQL correspondante

```
select distinct editeur_id from livres
```

Plusieurs conditions

On peut adjoindre la méthode `orWhere`.

Avec Eloquent

```
<?php
$livres=App\Livre::where('titre', '<', 'c')
->orWhere('titre', '>', 'v')
->get();
foreach ($livres as $livre) {
  echo $livre->titre, '<br>';
}
```

Avec le Query Builder

```
<?php
$livres=DB::table('livres')
->where('titre', '<', 'c')
->orWhere('titre', '>', 'v')
->get();
foreach ($livres as $livre) {
  echo $livre->titre, '<br>';
}
```

Requête SQL correspondante

```
select * from livres where titre<'c' or titre>'v'
```

Copyright © 2016 Eyrolles.

Encadrer des valeurs

Avec Eloquent

```
<?php
$livres=App\Livre::whereBetween('titre', array('g','k'))->get();
foreach ($livres as $livre) {
  echo $livre->titre, '<br>';
}
```

Avec le Query Builder

```
<?php
$livres=DB::table('livres')->whereBetween('titre', array('g','k'))->get();
foreach ($livres as $livre) {
  echo $livre->titre, '<br>';
}
```

Requête SQL correspondante

```
select * from livres where titre between 'g' and 'k'
```

On peut faire l'inverse avec `whereNotBetween` :

Avec Eloquent

```
<?php
$livres=App\Livre::whereNotBetween ('titre', array('g','k'))->get();
foreach ($livres as $livre) {
  echo $livre->titre, '<br>';
}
```

Avec le Query Builder

```
<?php
$livres=DB::table('livres')->whereNotBetween ('titre', array('g','k'))-
>get();
foreach ($livres as $livre) {
  echo $livre->titre, '<br>';
}
```

Requête SQL correspondante

```
select * from livres where titre not between 'g' and 'k'
```

Copyright © 2016 Eyrolles.

Prendre des valeurs dans un tableau

Avec Eloquent

```
<?php
$livres=App\Livre::whereIn('editeur_id', [10,11,12])->get();
foreach ($livres as $livre) {
  echo $livre->titre, '<br>';
}
```

Avec le Query Builder

```
<?php
$livres=DB::table('livres')->whereIn('editeur_id', [10,11,12])->get();
foreach ($livres as $livre) {
  echo $livre->titre, '<br>';
}
```

Requête SQL correspondante

```
select * from livres where editeur_id in ('10', '11', '12')
```

Ordonner et grouper

Avec Eloquent

```
<?php
$livres=App\Livre::
  select('editeur_id', DB::raw('count(id) as livre_count'))
  ->groupBy('editeur_id')
  ->get();
foreach ($livres as $livre) {
  echo $livre->editeur_id . ' => ' . $livre->livre_count, '<br>';
}
```

Ici, on groupe les lignes par éditeur et on compte les livres. La clause `select` mérite un petit commentaire. Quand on ne sait pas faire quelque chose avec le `Query Builder`, on a toujours la ressource d'utiliser une expression brute avec `DB::raw` comme je l'ai fait ici pour utiliser la fonction `count` de SQL.

Si vous utilisez une expression brute dans une requête, celle-ci n'est pas immunisée contre les injections SQL ; vous devez donc prendre vous-mêmes des mesures de sécurité.

Copyright © 2016 Eyrolles.

Avec le Query Builder

```
<?php
$livres=DB::table('livres')
  ->select('editeur_id', DB::raw('count(id) as livre_count'))
  ->groupBy('editeur_id')
  ->get();
foreach ($livres as $livre) {
  echo $livre->editeur_id . ' => ' . $livre->livre_count, '<br>';
}
```

Requête SQL correspondante

```
select editeur_id, count(id) as livre_count from livres group by editeur_id
```

Les jointures

Pour le moment, on a vu des requêtes qui ne concernent qu'une seule table, ce qui n'est pas le plus répandu. Lorsque deux tables sont concernées, on doit réaliser une jointure.

Trouver les titres des livres pour un éditeur dont on connaît l'identifiant

Avec Eloquent

```
<?php
$livres=App\Editeur::find(11)->livres;
foreach ($livres as $livre) {
  echo $livre->titre, '<br>';
}
```

Avec le Query Builder

```
<?php
$livres=DB::table('editeurs')
->where('editeurs.id', 11)
->join('livres', 'livres.editeur_id', '=', 'editeurs.id')
->get();
foreach ($livres as $livre) {
  echo $livre->titre, '<br>';
}
```

La jointure s'effectue avec la méthode `join`. Les requêtes générées ne sont pas les mêmes dans les deux cas.

Copyright © 2016 Eyrolles.

Requêtes SQL correspondantes avec Eloquent

```
select * from editeurs where id='11' limit 1
select * from livres where livres.editeur_id='11'
```

Requête SQL correspondante avec le Query Builder

```
select * from editeurs inner join livres on livres.editeur_id=editeurs.id
where editeurs.id='11'
```

Eloquent, dans ce cas, ne fait pas de jointure.

Trouver les livres d'un auteur dont on connaît le nom

Avec Eloquent

```
<?php
$livres=App\Livre::whereHas('auteurs', function($q)
{
  $q->whereNom('Osbaldo White');
})->get();
foreach ($livres as $livre) {
  echo $livre->titre, '<br>';
}
```

Requête SQL correspondante

```
select * from livres where (select count(*) from auteurs inner join
auteur_livre on auteurs.id=auteur_livre.auteur_id where auteur_livre.livre_
id=livres.id and nom='Osbaldo White')>='1'
```

La requête est un peu acrobatique, mais elle fonctionne.

Avec le Query Builder

```
<?php
$livres=DB::table('livres')
->join('auteur_livre', 'livres.id', '=', 'auteur_livre.livre_id')
->join('auteurs', 'auteurs.id', '=', 'auteur_livre.auteur_id')
->where('auteurs.nom', '=', 'Osbaldo White')
->get();
foreach ($livres as $livre) {
  echo $livre->titre, '<br>';
}
```

Il réalise ici une double jointure.

Copyright © 2016 Eyrolles.

Requête SQL correspondante

```
select * from livre inner join auteur_livre on livres.id=auteur_livre.livre_
id inner join auteurs on auteurs.id=auteur_livre.auteur_id where auteurs.
nom='Osbaldo White'
```

Trouver les auteurs pour un éditeur dont on connaît l'identifiant

Avec Eloquent

```
<?php
$livres=App\Editeur::find(10)->livres;
foreach ($livres as $livre) {
  foreach($livre->auteurs as $auteur) {
    echo $auteur->nom, '<br>';
  }
}
```

On trouve facilement les livres de l'éditeur, mais pour aller jusqu'aux auteurs il faut boucler sur les résultats.

Requêtes SQL correspondantes

```
select * from editeurs where id='10' limit 1
select * from livres where livres.editeur_id='10'
select auteurs.*, auteur_livre.livre_id as pivot_livre_id, auteur_livre.
auteur_id as pivot_auteur_id from auteurs inner join auteur_livre
on auteurs.id=auteur_livre.auteur_id where auteur_livre.livre_id='28'
```

Avec le Query Builder

```
<?php
$auteurs=DB::table('auteurs')
->select('auteurs.nom')
->join('auteur_livre', 'auteurs.id', '=', 'auteur_livre.auteur_id')
->join('livres', 'livres.id', '=', 'auteur_livre.livre_id')
->join('editeurs', function($join)
  {
    $join->on('editeurs.id', '=', 'livres.editeur_id')
    ->where('editeurs.id', '=', 10);
  })
->get();
foreach($auteurs as $auteur) {
  echo $auteur->nom, '<br>';
}
```

C'est évidemment plus compact car il n'y a qu'une seule requête.

Copyright © 2016 Eyrolles.

Requête SQL correspondante

```
select auteurs.nom from auteurs inner join auteur_livre on auteurs.id =
auteur_livre.auteur_id inner join livres on livres.id = auteur_livre.livre_
id inner join editeurs on editeurs.id = livres.editeur_id and editeurs.id =
'10'
```

Attention aux requêtes imbriquées

Il faut être prudent dans certaines situations. Par exemple, supposez que vous vouliez avoir la liste des auteurs avec pour chacun la liste de ses ouvrages. Vous pourriez écrire un code semblable au suivant :

Avec Eloquent

```
<?php
$auteurs=App\Auteur::all();
foreach ($auteurs as $auteur) {
  echo '<h1>'. $auteur->nom .'</h1>';
  foreach($auteur->livres as $livre) {
    echo $livre->titre, '<br>';
  }
}
```

Cela fonctionne très bien, mais si vous regardez vos requêtes, vous allez être effrayés ! Dans mon cas, j'en trouve 41, tout simplement parce que, pour chaque auteur, une requête est lancée pour trouver ses livres.

Dans ce genre de situation, il faut absolument utiliser le chargement lié, c'est-à-dire demander à `Eloquent` de charger la table `livres` avec la méthode `with` :

```
<?php
$auteurs=App\Auteur::with('livres')->get();
foreach ($auteurs as $auteur) {
  echo '<h1>'. $auteur->nom .'</h1>';
  foreach($auteur->livres as $livre) {
    echo $livre->titre, '<br>';
  }
}
```

Le changement peut paraître minime, mais on n'a plus que deux requêtes.

Requêtes SQL correspondantes

```
select * from auteurs
select livres.*, auteur_livre.auteur_id as pivot_auteur_id, auteur_
livre.livre_id as pivot_livre_id from livres inner join auteur_livre
on livres.id=auteur_livre.livre_id where auteur_livre.auteur_id
in ('1','2','3','4','5','6','7','8','9','10','11','12','13','14','15','16',
'17','18','19','20','21','22','23','24','25','26','27','28','29','30','31',
'32','33','34','35','36','37','38','39','40')
```

Copyright © 2016 Eyrolles.

Comment faire la même chose avec le `Query Builder` ? Ce n'est pas si simple ; voici une solution avec utilisation d'une expression brute :

Avec le Query Builder

```
<?php
$results=DB::table('auteurs')
->select('nom', DB::raw('group_concat(titre) as titres'))
->groupBy('nom')
->join('auteur_livre', 'auteurs.id', '=', 'auteur_livre.auteur_id')
->join('livres', 'livres.id', '=', 'auteur_livre.livre_id')
->get();
foreach ($results as $result) {
  echo '<h1>'. $result->nom .'</h1>';
  $titres=explode(',', $result->titres);
  foreach($titres as $titre) {
    echo $titre, '<br>';
  }
}
```

On n'a désormais plus qu'une seule requête.

Requête SQL correspondante

```
select nom, group_concat(titre) as titres from auteurs inner join auteur_
livre on auteurs.id=auteur_livre.auteur_id inner join livres on livres.
id=auteur_livre.livre_id group by nom
```

Vous n'arriverez pas toujours à réaliser ce que vous désirez uniquement avec `Eloquent` ; il vous faudra alors utiliser le `Query Builder`.

Ce chapitre est loin d'épuiser le sujet du `Query Builder`. Vous pourrez trouver tous les compléments utiles sur *http://laravel.com/docs/5.2/queries*.

En résumé

- `Eloquent` permet de faire beaucoup de manipulations sur les tables et est à l'aise avec les relations.
- Le `Query Builder` est le complément indispensable d'`Eloquent`.
- Parfois, il est plus efficace d'utiliser le `Query Builder` qu'`Eloquent`.
- Dans certains cas, on doit utiliser des expressions brutes dans les requêtes, mais il faut alors penser à se protéger des injections SQL.

Copyright © 2016 Eyrolles.

Troisième partie

Plus loin avec Laravel

Il est certes assez pratique de développer un site sur un serveur local, mais il arrive toujours un moment où il faut procéder au déploiement sur le serveur de production. Nous verrons comment procéder avec une application Laravel.

Le code PHP avec Laravel est en général bien structuré et organisé, et il est aussi très lisible. On ne peut pas dire qu'il en est toujours ainsi au niveau des vues qui peuvent devenir très chargées et difficiles à lire. Je vous propose donc des stratégies pour les rendre plus simples et bien organisées.

Nous verrons aussi dans cette partie comment localiser un site avec Laravel, c'est-à-dire le rendre adapté au langage de l'utilisateur.

J'aborderai également la mise en œuvre d'Ajax et la gestion des autorisations et des événements.

Enfin, nous aborderons la question des tests unitaires qui constituent une phase importante du développement.

Copyright © 2016 Eyrolles.

Copyright © 2016 Eyro les.

18 Environnement et déploiement

Une application se développe et se teste en local, mais il arrive un moment où il faut la mettre sur un serveur pour qu'elle devienne visible et accessible. Ce déploiement n'est pas forcément une tâche aisée selon le contexte. Dans ce chapitre, nous allons faire un petit tour d'horizon de ce qu'il convient de faire, sans pouvoir être exhaustif en raison des multiples configurations existantes.

L'environnement

La version 5 de Laravel a révolutionné les habitudes au niveau de l'environnement, en comparaison à la version 4. Elle fait désormais usage d'un package tiers : `DotEnv` (*https://github.com/vlucas/phpdotenv*). On y gagne en clarté, mais pas forcément en ergonomie.

Quand vous installez Laravel avec `Composer`, vous trouvez à la racine le fichier `.env` avec le contenu suivant :

```
APP_ENV=local
APP_DEBUG=true
APP_KEY=oJiCzJoPu9FvTjmTz1jpsNHVSBL7i1nl
APP_URL=http://localhost

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret

CACHE_DRIVER=file
SESSION_DRIVER=file
QUEUE_DRIVER=sync
```

Copyright © 2016 Eyrolles.

```
REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_DRIVER=smtp
MAIL_HOST=mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

Les informations se présentent sous la forme clé/valeur. Par exemple, on a déjà vu cela en oeuvre dans `config/app.php` :

```
<?php
'debug'=>env('APP_DEBUG', false),
```

L'*helper* `env` lit la valeur de `APP_DEBUG` dans le fichier `.env` et l'affecte à la clé `debug` dans la configuration de l'application. Remarquez qu'il accepte une valeur par défaut comme deuxième paramètre.

On trouve aussi par exemple le réglage de MySQL :

```
<?php
'host'=>env('DB_HOST', 'localhost'),
'database'=>env('DB_DATABASE', 'forge'),
'username'=>env('DB_USERNAME', 'forge'),
'password'=>env('DB_PASSWORD', ''),
```

Cet *helper* stocke les valeurs dans la super globale d'environnement `$_ENV` du serveur.

Moralité : pour disposer de plusieurs environnements, il faut créer plusieurs fichiers `.env`. Le cas le plus classique sera d'avoir un environnement local de développement et un autre de déploiement. Il faut juste ne pas se mélanger les pinceaux, en particulier ne pas enlever le fichier `.gitignore` de la racine, qui prévoit de ne pas envoyer le fichier `.env` susceptible de contenir des données sensibles (si vous utilisez `git`).

La variable `APP_ENV` dans le fichier `.env` est justement destinée à connaître l'environnement actuel :

```
APP_ENV=local
```

Lorsque je crée une application, je copie le fichier d'environnement pour régler mes variables en situation de production. Pour qu'il n'y ait pas de confusion, je le nomme de façon explicite.

Copyright © 2016 Eyrolles.

Le fichier d'environnement de production

Dans ce fichier, je fixe toutes les valeurs nécessaires :

```
APP_ENV=production
APP_DEBUG=false
APP_KEY=5avz0M3IGPu4GFxBBLPhQs00MNJREzoL
...
```

Il me suffit ensuite d'envoyer ce fichier-là sur le serveur et de le renommer.

Selon l'endroit où vous hébergez votre application, vous avez parfois la possibilité de définir les variables d'environnement sans utiliser le fichier `.env`. C'est le cas si vous utilisez par exemple Forge (*https://forge.laravel.com/*).

Le déploiement

Nous allons envisager plusieurs situations pour le déploiement selon votre hébergement.

Besoins de Laravel

Laravel nécessite :

- PHP ≥ 5.5.9 ;
- extension PHP `PDO` ;
- extension PHP `OpenSSL` ;
- extension PHP `Mbstring` ;
- extension PHP `Tokenizer`.

Par ailleurs, le dossier `storage` doit avoir les droits d'écriture sur le serveur.

Selon la méthode d'installation que vous allez employer, vérifiez que vous avez bien une clé de cryptage dans votre environnement.

Copyright © 2016 Eyrolles.

Solution royale

Commençons par la situation idéale : vous disposez de SSH et vous avez un dossier disponible avec possibilité de définir des dossiers non accessibles.

Si vous disposez de `git` sur le serveur, vous pouvez évidemment l'utiliser et vous n'aurez aucun souci.

Si vous avez `Composer` installé globalement sur le serveur, c'est aussi parfait. Vous pouvez alors installer l'installeur de Laravel :

```
composer global require "laravel/installer"
```

Il ne vous reste plus alors qu'à utiliser cet installeur :

```
laravel new application
```

Malheureusement, il est probable que vous ne disposiez pas de `Composer`. Dans ce cas, commencez par envoyer son archive `phar` (*https://getcomposer.org/download/*). Vous accédez à votre dossier avec SSH :

```
login as: mon_login
login@login.org's password:*****
```

Ensuite, lancez la creation avec `Composer` :

```
php composer create-project --prefer-dist laravel/laravel
```

Attendez quelques minutes que toutes les dépendances s'installent.

Cela devrait fonctionner mais avec un Laravel anonyme qui n'est pas encore une application.

Laravel installé

Il faut ensuite envoyer le dossier `app`, le `composer.json`, le fichier `.env`, créer une base de données… tout ce qui spécifie une application.

Avec SSH, il existe des applications de déploiement automatisé comme `envoy` (*https://github.com/laravel/envoy*), `capistrano` (*https://github.com/capistrano/capistrano*) ou `magallanes` (*https://github.com/andres-montanez/Magallanes*).

Copyright © 2016 Eyrolles.

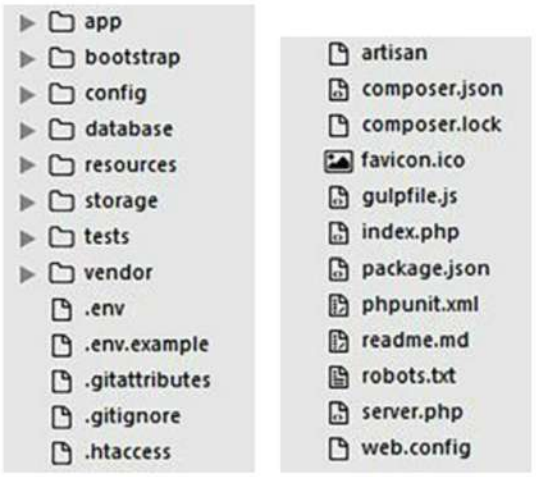
Solution intermédiaire

Supposons maintenant que vous disposiez de SSH, mais que vous soyez obligé de rendre votre application complètement accessible. Dans ce cas, vous devez supprimer le dossier `public` et effectuer quelques manipulations pour que cela fonctionne. Voici la procédure.

1. Commencez par installer normalement Laravel. Vous devez avoir l'architecture suivante.

L'architecture de Laravel

2. Transférez à la racine tout ce qui se trouve dans `public` et supprimez ce dossier.



Le dossier public a été supprimé et son contenu transféré.

Évidemment, plus rien ne fonctionne !

3. Ouvrez le fichier `index.php` et trouvez la ligne suivante :

```
<?php
require __DIR__.'/../bootstrap/autoload.php';
```

Copyright © 2016 Eyrolles.

Remplacez-la par celle-ci :

```
<?php
require __DIR__.'/bootstrap/autoload.php';
```

4. Dans le même fichier, trouvez la ligne :

```
<?php
$app=require_once __DIR__.'/../bootstrap/app.php';
```

Remplacez-la par celle-ci :

```
<?php
$app=require_once __DIR__.'/bootstrap/app.php';
```

La page d'accueil de Laravel s'affiche désormais correctement.

5. L'inconvénient, c'est que désormais tout devient accessible, il faut donc prendre des précautions, par exemple prévoir dans les dossiers à protéger (`app`, `bootstrap`...) un `.htaccess` contenant la ligne suivante :

```
Deny from all
```

Les seuls fichiers accessibles restent ceux qui sont présents à la racine.

Solution laborieuse

Si vous ne disposez pas de SSH, l'affaire devient plus laborieuse parce que la seule solution qui vous reste est le FTP. Comme il y a beaucoup de fichiers, le transfert peut être long selon la vitesse de votre transmission.

De plus, les mises à jour doivent être soigneusement gérées parce qu'elles vont nécessairement passer par le FTP.

Mode maintenance

Lors de la mise à jour de l'application, il peut y avoir des moments où il est préférable que personne ne se connecte. Laravel propose un mode *maintenance* facile à mettre en œuvre avec `Artisan` :

```
php artisan down
```

Bien sûr, il y a la commande inverse :

```
php artisan up
```


Copyright © 2016 Eyrolles.

Évidemment, encore faut-il que vous ayez accès à `Artisan` sur votre serveur…

En mode *maintenance*, la vue affichée est *resources/views/errors/503.blade.php*.

En résumé

- Il faut créer des environnements pour répondre aux différentes situations : local, distant…
- Le déploiement est facile et rapide si on dispose de `git` ou du SSH.
- On peut supprimer le dossier `public` tout en gardant un Laravel fonctionnel ; il faut juste prendre des mesures pour assurer la sécurité.
- Il est possible de déployer une application avec le FTP, mais c'est un peu laborieux et les mises à jour doivent être soigneusement gérées.

Copyright © 2016 Eyrolles.

Copyright © 2016 Eyro les.

19 Des vues propres

On a vu comment créer des classes bien organisées en se contentant d'effectuer leur tâche. En revanche, au niveau des vues c'est une autre histoire. En général, on utilise un framework CSS, par exemple Bootstrap dont je me suis servi dans les exemples de ce chapitre. Mais que se passe-t-il le jour où ce framework évolue ou si on décide d'en changer ?

Les macros

Problème

Reprenez la vue que nous avons créée dans l'exemple du blog personnel pour la création d'un article (`resources/views/posts/add.blade.php`) :

```
@extends('template')

@section('contenu')
  <br>
  <div class="col-sm-offset-3 col-sm-6">
    <div class="panel panel-info">
      <div class="panel-heading">Ajout d'un article</div>
      <div class="panel-body">
        {!! Form::open(['route'=>'post.store']) !!}
          <div class="form-group {!! $errors->has('titre') ? 'has-
error' : '' !!}">
           {!! Form::text('titre', null, ['class'=>'form-control',
'placeholder'=>'Titre']) !!}
            {!! $errors->first('titre', '<small class="help-block">:message</
small>') !!}
          </div>
```

Copyright © 2016 Eyrolles.

```
          <div class="form-group {!! $errors->has('contenu') ? 'has-
error' : '' !!}">
            {!! Form::textarea ('contenu', null, ['class'=>'form-control',
'placeholder'=>'Contenu']) !!}
            {!! $errors->first('contenu', '<small class="help-
block">:message</small>') !!}
          </div>
          <div class="form-group {{ $errors->has('tags') ? 'has-
error' : '' }}">
            {!! Form::text('tags', null, array('class'=>'form-control',
'placeholder'=>'Entrez les mots-clés séparés par des virgules')) !!}
            {!! $errors->first('tags', '<small class="help-block">:message</
small>') !!}
          </div>
          {!! Form::submit('Envoyer !', ['class'=>'btn btn-info pull-
right']) !!}
        {!! Form::close() !!}
      </div>
    </div>
    <a href="javascript:history.back()" class="btn btn-primary">
      <span class="glyphicon glyphicon-circle-arrow-left"></span>Retour
    </a>
  </div>
@endsection
```

Quel est le degré de dépendance entre Bootstrap et le code précédent ?

- mise en page avec la grille ;
- utilisation du composant `panel` ;
- utilisation des classes `btn`, `btn-info`, `form-control`, `form-group`… pour le formulaire ;
- utilisation de la classe `has-error` pour le retour de validation ;
- …

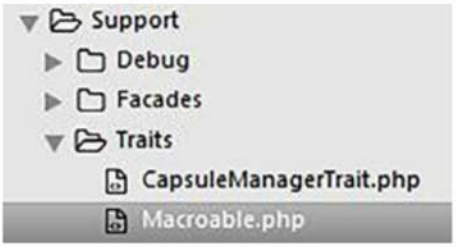
Si je décide de changer de framework, je dois tout recommencer. Si je n'ai qu'un formulaire de ce genre, ce n'est pas trop grave, mais si j'en ai 10 ou 20, cela devient embarrassant. De même, si le framework évolue (ce qui est fréquent par exemple avec Bootstrap) et si je désire suivre cette évolution, je serai confronté au même problème.

L'idéal est de pouvoir construire ma vue sans aucun lien avec un framework, avec des méthodes neutres. Il y a plusieurs façons de le faire, mais la plus simple est certainement l'utilisation de macros.

Définition

Une macro est un outil de substitution : on utilise un texte et cela en produit un autre avec possibilité de passer des paramètres. Laravel permet la création de macros. Regardez dans les dossiers du framework pour trouver le fichier suivant.

Copyright © 2016 Eyrolles.



Le fichier pour les macros

Regardons le code ; vous constatez qu'il s'agit d'un `trait` qui expose quelques méthodes, dont :

```
<?php
public static function macro($name, callable $macro)
{
    static::$macros[$name]=$macro;
}
```

En utilisant ce `trait`, on peut donc enregistrer des macros dans la propriété statique `$macros`. Certaines méthodes magiques permettent aussi d'utiliser les macros mémorisées.

Il se trouve que le composant `LaravelCollective\Html` utilise ce `trait`. Par exemple, dans la classe `FormBuilder`, on trouve :

```
<?php
class FormBuilder {

    use Macroable, Componentable {
        Macroable::__call as macroCall;
        Componentable::__call as componentCall;
    }
```

Il en est de même pour la classe `HtmlBuilder`. Nous allons donc utiliser cette possibilité pour rendre notre vue plus propre.

On veut obtenir une vue conçue comme suit :

```
@extends('template')

@section('contenu')
  <br>
 <div class="col-sm-offset-3 col-sm-6">
    <div class="panel panel-info">
     <div class="panel-heading">Ajout d'un article</div>
      <div class="panel-body">
       {!! Form::open(['route'=>'post.store']) !!}
          {!! Form::control('text', $errors, 'titre', 'Titre') !!}
```

Copyright © 2016 Eyrolles.

```
            {!! Form::control('textarea', $errors, 'contenu', 'Contenu') !!}
            {!! Form::control('text', $errors, 'tags', 'Entrez les mots-clés
séparés par des virgules') !!}
            {!! Form::button_submit('Envoyer !') !!}
          {!! Form::close() !!}
        </div>
      </div>
      {!! Html::button_back() !!}
    </div>
@endsection
```

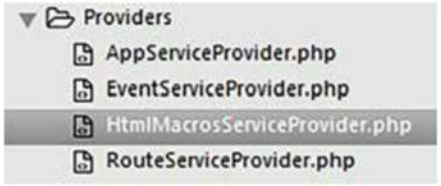
Nous n'avons plus aucune trace de notre framework au niveau du formulaire.

Maintenant la question est : où mettre les macros ?

Fournisseur de services

Un fournisseur (`provider`) est le lieu idéal pour enregistrer des services. Laravel comporte de nombreux fournisseurs. Nous allons en créer un pour nos macros. Une fois de plus, `Artisan` nous facilite la tâche :

```
php artisan make:provider HtmlMacrosServiceProvider
```



Le fournisseur pour les macros

Voici son code :

```
<?php
namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class HtmlMacrosServiceProvider extends ServiceProvider
{
  /**
   * Bootstrap the application services.
   *
   * @return void
   */
 public function boot()
  {
    //
 }
```

Copyright © 2016 Eyrolles.

```
    /**
     * Register the application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

La méthode `register` permet d'enregistrer tout ce dont on a besoin, notamment nos macros.

La méthode `boot` est exécutée une fois que tous les fournisseurs de l'application ont été enregistrés, ce qui permet alors de disposer de tous les services.

Ajoutons nos macros :

```
<?php
namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Collective\Html\FormBuilder;
use Collective\Html\HtmlBuilder;

class HtmlMacrosServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->registerFormControl();
        $this->registerFormSubmit();
        $this->registerHtmlButtonBack();
    }

    private function registerFormControl()
    {
        FormBuilder::macro('control', function($type, $errors, $nom,
$placeholder)
        {
            $valeur=\Request::old($nom) ? \Request::old($nom):null;
            $attributes=['class'=>'form-control', 'placeholder'=>$placeholder];
            return sprintf('
                <div class="form-group %s">
                    %s
                    %s
                </div>',
                $errors->has($nom) ? 'has-error' : '',
                call_user_func_array(['Form', $type], [$nom, $valeur, $attributes]),
                $errors->first($nom, '<small class="help-block">:message</small>')
            );
        });
```

Copyright © 2016 Eyrolles.

```
    }

    private function registerFormSubmit()
    {
        FormBuilder::macro('button_submit', function($texte)
        {
            return FormBuilder::submit($texte, ['class'=>'btn btn-info pull-
right']);
        });
    }

    private function registerHtmlButtonBack()
    {
        HtmlBuilder::macro('button_back', function()
        {
            return '<a href="javascript:history.back()" class="btn btn-primary">
                        <span class="glyphicon glyphicon-circle-arrow-left"></
span>Retour
                    </a>';
        });
    }
}
```

Ce n'est qu'un exemple et le codage pourrait être différent, c'est le principe qui importe. On pourrait aussi localiser ailleurs les macros et se contenter de les appeler à partir du fournisseur. C'est alors juste une question d'organisation du code. Il est évident que les fournisseurs ne sont pas prévus pour accueillir un code volumineux, comme toute classe d'ailleurs. Ici, on se contentera de cette présentation.

Il ne reste plus qu'à informer Laravel que notre fournisseur existe dans `config/app.php` :

```
<?php
/*
 * Application Service Providers...
 */
App\Providers\AppServiceProvider::class,
App\Providers\AuthServiceProvider::class,
App\Providers\EventServiceProvider::class,
App\Providers\RouteServiceProvider::class,
Collective\Html\HtmlServiceProvider::class,
App\Providers\HtmlMacrosServiceProvider::class,
```

Normalement, le formulaire de création d'un article devrait encore fonctionner.

Copyright © 2016 Eyrolles.

Le formulaire généré par les macros

On a ainsi pu assainir la vue au niveau du formulaire.

Les templates

Il reste encore des éléments du framework dans notre vue : la grille et le `panel`. dans ce cas, une macro ne serait pas vraiment adaptée. Il est préférable de créer un template pour ce formulaire, qui pourra également servir pour d'autres formulaires (`app/views/template_form.blade.php`) :

```
@extends('template')

@section('contenu')
  <br>
  <div class="col-sm-offset-3 col-sm-6">
    <div class="panel panel-info">
      <div class="panel-heading">
        @yield('titre')
      </div>
      <div class="panel-body">
        @yield('formulaire')
      </div>
    </div>
    {!! Html::button_back() !!}
  </div>
@endsection
```

Je vous rappelle le contenu du template principal (`app/views/template.blade.php`) :

Copyright © 2016 Eyrolles.

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Mon joli site</title>
    {!! HTML::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/
bootstrap.min.css') !!}
    {!! HTML::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/
bootstrap-theme.min.css') !!}
    <!--[if lt IE 9]>
      {{ HTML::style('https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.
js') }}
      {{ HTML::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.
min.js') }}
    <![endif]-->
    <style>textarea {resize:none;}</style>
  </head>
  <body>
    <header class="jumbotron">
      <div class="container">
        <h1 class="page-header">
          {!! link_to_route('post.index', 'Mon joli blog') !!}
        </h1>
        @yield('header')
      </div>
    </header>
    <div class="container">
      @yield('contenu')
    </div>
  </body>
</html>
```

On peut obtenir une vue très épurée pour notre formulaire :

```
@extends('template_form')

@section('titre')
    Ajout d'un article
@endsection

@section('formulaire')
  {!! Form::open(['route'=>'post.store']) !!}
    {!! Form::control('text', $errors, 'titre', 'Titre') !!}
    {!! Form::control('textarea', $errors, 'contenu', 'Contenu') !!}
    {!! Form::control('text', $errors, 'tags', 'Entrez les mots-clés séparés
par des virgules') !!}
    {!! Form::button_submit('Envoyer !') !!}
  {!! Form::close() !!}
@endsection
```

Copyright © 2016 Eyrolles.

On ne retrouve plus aucune trace de framework. Quelles que soient les modifications d'interface ultérieures, cette vue n'aura pas à être modifiée. Si on change de framework, on a juste le template général et celui des formulaires à modifier.

Vous devez bien organiser vos templates pour que votre code soit efficace, ce qui dépend évidemment de votre application.

Si vous avez beaucoup de vues, il est judicieux de créer des dossiers pour les classer. Par exemple, vous pouvez créer un dossier `templates`.

Une autre organisation du code peut amener à créer des vues partielles à insérer dans la vue finale, on parle ainsi de `partials`. C'est très utilisé par exemple pour les barres de message d'erreur. Il existe ainsi de nombreuses façons d'organiser les vues, l'essentiel étant d'aboutir à un code simple, lisible et facile à maintenir.

En résumé

- L'utilisation de macros rend les vues indépendantes du framework utilisé.
- Un fournisseur de services permet de faire des initialisations.
- La création de templates aide à bien organiser le code des vues.
- L'utilisation de dossiers est souvent nécessaire lorsqu'il y a beaucoup de vues à classer.

Copyright © 2016 Eyrolles.

Copyright © 2016 Eyro les.

20 Des vues propres avec le conteneur de dépendances

Nous allons continuer notre démarche de nettoyage des vues, qui nous sert de prétexte pour explorer le conteneur de dépendances constituant le cœur de Laravel. Ce conteneur est une sorte de boîte dans laquelle on peut mettre plein de choses. C'est notamment lui qui facilite la gestion des injections de dépendances. Nous l'avons déjà utilisé pour gérer la résolution des dépendances. Dans ce chapitre, je poursuivrai l'exemple des macros du chapitre précédent, en utilisant cette fois une classe.

La nouvelle vue

Dans le chapitre précédent, on a séparé le code pour couper un article en deux : un template (`resources/views/template_form.blade.php`) et le formulaire épuré grâce aux macros (`resources/views/post/add.blade.php`).

Je vous propose de remplacer les macros par des méthodes d'une classe dédiée et d'ajouter le traitement du panneau. Le but est de se passer du template et d'aboutir à la vue suivante :

```
@extends('template')

@section('contenu')
  <br>
  <div class="col-sm-offset-3 col-sm-6">
    {!! Panel::
      head(
        'Ajout d\'un article'
      )
      ->body(
        Form::open(['route'=>'post.store']).
          Form::control('text', $errors, 'titre', 'Titre').
          Form::control('textarea', $errors, 'contenu', 'Contenu').
```

Copyright © 2016 Eyrolles.

```
            Form::control('text', $errors, 'tags', 'Entrez les mots-clés
séparés par des virgules').
            Form::button_submit('Envoyer !').
          Form::close()
        )
        ->type('primary')
      !!}
      {!! Html::button_back() !!}
    </div>
@endsection
```

On ne voit pas la différence pour le formulaire lui-même puisqu'elle se situe au niveau de l'intendance. En revanche, une nouvelle classe (`Panel`) apparaît avec des méthodes pour construire le panneau.

Évidemment, l'aspect de la vue, lui, ne doit pas changer.

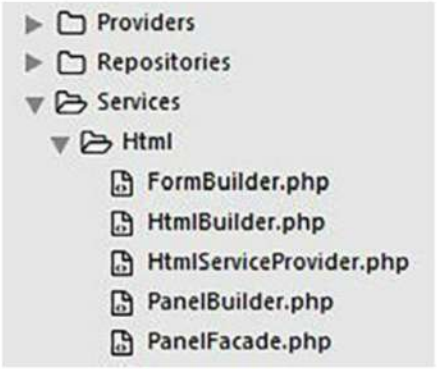
La vue de création d'un article

L'organisation du code

Dans le chapitre précédent, on a écrit le code des macros dans le fournisseur de services. C'était simple et efficace. Cette solution est envisageable lorsque le code n'est pas trop fourni ou complexe ; dans le cas contraire, il devient vite indispensable de le répartir dans plusieurs autres classes.

La notion de « service » est suffisamment générale pour inclure la plupart des tâches à réaliser dans une application. Je vous propose de créer un service pour gérer ces fonctionnalités pour les vues. On créera les dossiers `Services` et `Html` pour ranger tout le code dont nous aurons besoin pour ce chapitre.

Copyright © 2016 Eyrolles.



L'organisation des fichiers

- `FormBuilder` contiendra les méthodes `control` et `button_submit`, les deux qui concernent le formulaire.
- `HtmlBuilder` contiendra la méthode `button_back`.
- `HtmlServiceProvider` est le fournisseur qui initialisera tous les composants.
- `PanelBuilder` contiendra les méthodes pour le panneau.
- `PanelFacade` générera la façade pour le panneau. On pourra ainsi utiliser une syntaxe simplifiée.

Les constructeurs

FormBuilder et HtmlBuilder

Pour créer ces deux constructeurs, il faut juste reprendre le code des macros et l'écrire dans deux classes séparées. Pour les deux méthodes du formulaire, on aura donc la classe `FormBuilder` :

```
<?php
namespace App\Services\Html;

use Collective\Html\FormBuilder as CollectiveFormbuilder;
use Request;

class FormBuilder extends CollectiveFormbuilder
{
 public function control($type, $errors, $nom, $placeholder)
  {
   $valeur=Request::old($nom) ? Request::old($nom):null;
   $attributes=['class'=>'form-control', 'placeholder'=>$placeholder];
   return sprintf('
      <div class="form-group %s">
       %s
        %s
     </div>',
```

Copyright © 2016 Eyrolles.

```
      $errors->has($nom) ? 'has-error' : '',
      call_user_func_array(['Form', $type], [$nom, $valeur, $attributes]),
      $errors->first($nom, '<small class="help-block">:message</small>')
    );
  }

  public function button_submit($texte)
  {
    return parent::submit($texte, ['class'=>'btn btn-info pull-right']);
  }
}
```

Vous remarquez que je me contente d'étendre la classe `FormBuilder` du package pour en conserver toutes les méthodes.

Et voici ce que devient le HTML dans la classe `HtmlBuilder` :

```
<?php
namespace App\Services\Html;

use Collective\Html\HtmlBuilder as CollectiveHtmlBuilder;

class HtmlBuilder extends CollectiveHtmlBuilder
{
  public function button_back()
  {
    return '<a href="javascript:history.back()" class="btn btn-primary">
      <span class="glyphicon glyphicon-circle-arrow-left"></span>Retour
    </a>';
  }
}
```

Ici aussi, je me contente d'étendre la classe du package pour ajouter ma méthode.

PanelBuilder

Pour le panneau, on créera aussi une classe (`PanelBuilder`) :

```
<?php
namespace App\Services\Html;

class PanelBuilder
{
  protected $type;
 protected $head;
 protected $body;
 protected $foot;

  public function __construct($type='default', $head=null, $body=null,
$foot=null)
 {
    $this->type=$type;
```

Copyright © 2016 Eyrolles.

```
    $this->head=$head;
    $this->body=$body;
    $this->foot=$foot;
  }

 public function __toString()
  {
    $s='<div class="panel panel-'.$this->type.'">';
    if ($this->head)
    {
      $s.='<div class="panel-heading">'.$this->head.'</div>';
    }
    if ($this->body)
    {
      $s.='<div class="panel-body">'.$this->body.'</div>';
    }
    if ($this->foot)
    {
      $s.='<div class="panel-footer">'.$this->foot.'</div>';
    }
    $s.="</div>";
    return $s;
  }

 public function type($type)
  {
    $this->type=$type;
    return $this;
  }

  public function head($head)
  {
    $this->head=$head;
    return $this;
  }

  public function body($body)
  {
    $this->body=$body;
    return $this;
  }

  public function footer($foot)
  {
    $this->foot=$foot;
    return $this;
  }
}
```

J'ai prévu de pouvoir transmettre les paramètres soit dans le constructeur, soit au moyen de méthode dédiées pour généraliser le code.

Il faut aussi créer la façade pour la classe `Panel` :

Copyright © 2016 Eyrolles.

```
<?php
namespace App\Services\Html;

use Illuminate\Support\Facades\Facade;

class PanelFacade extends Facade
{
  protected static function getFacadeAccessor() {return 'panel';}
}
```

La syntaxe d'une façade est toute simple ; ici, on renvoie juste la référence `'panel'`. Nous allons voir bientôt d'où sort cette référence et comment on la nomme.

Fournisseur de services et façade

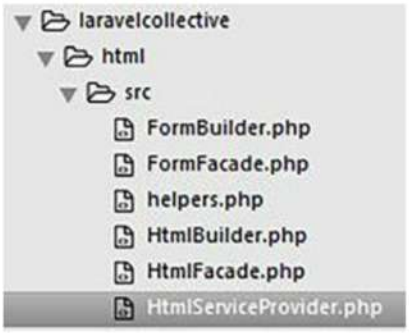
C'est dans le fournisseur que va se situer l'intendance pour le framework. En voici le code :

```
<?php
namespace App\Services\Html;

use Illuminate\Support\ServiceProvider;

class HtmlServiceProvider extends ServiceProvider
{
  public function register()
  {
    $this->registerHtmlBuilder();
    $this->registerFormBuilder();
    $this->registerPanelBuilder();
  }

 protected function registerHtmlBuilder()
  {
    $this->app->singleton('html', function ($app) {
      return new HtmlBuilder($app['url'], $app['view']);
    });
  }

 protected function registerFormBuilder()
  {
    $this->app->singleton('form', function ($app) {
      $form=new FormBuilder($app['html'], $app['url'], $app['view'],
$app['session.store']->getToken());
     return $form->setSessionStore($app['session.store']);
    });
 }
 protected function registerPanelBuilder()
  {
    $this->app->singleton('panel', function()
   {
```

Copyright © 2016 Eyrolles.

```
        return new PanelBuilder;
    });
  }
}
```

Je vous ai déjà dit que les fournisseurs servent à enregistrer un certain nombre de choses : des liaisons de dépendance, des événements, des éléments de configuration. Ici, nous enregistrons les méthodes pour les formulaires et le HTML. Il existe déjà un fournisseur dans le package, mais je suis obligé de surcharger la plupart de ses méthodes.



Le fournisseur du package

La méthode la plus importante d'un fournisseur est `register`. Lorsqu'une requête est prise en charge par Laravel, on a besoin de tout un tas de choses : configuration des erreurs, détection de l'environnement, définition des *middlewares* que la requête devra traverser, vérification du CSRF… À un moment, on passera aussi en revue les fournisseurs et exécuter leur méthode `register`, les uns après les autres dans leur ordre d'inscription. On en profite pour déclarer toutes les dépendances dans le conteneur.

Le conteneur est un peu le grand sac de l'application qui permet de tout ranger et de tout retrouver. Voici l'exemple d'une méthode :

```
<?php
protected function registerPanelBuilder()
{
    $this->app->singleton('panel', function()
    {
        return new PanelBuilder;
    });
}
```

On dit au conteneur : « Tu vas te souvenir de `'panel'` et, si j'en ai besoin, j'aurai juste à te dire `'panel'` et tu me renverras une instance de `PanelBuilder`. » Le conteneur créera donc un lien (`singleton`) entre la référence `'panel'` et la classe `PanelBuilder`.

Copyright © 2016 Eyrolles.

Si on utilise `singleton`, c'est qu'on veut une unique résolution ; sinon, on utiliserait `bind`.

Maintenant, vous comprenez mieux le code de la façade :

```
<?php
class PanelFacade extends Facade
{
  protected static function getFacadeAccessor() {return 'panel';}
}
```

Là, on dit que si on rencontre la façade `Panel`, on doit aller chercher dans le conteneur l'instance de la classe qui correspond à `'panel'`. Comme on a pris soin dans le fournisseur de créer un lien entre cette référence et le nom de la classe, cela va fonctionner !

```
<?php
Panel::head()
```

Dans cet exemple, Laravel comprendra que je parle de la façade, que je veux une instance de la classe `PanelBuilder` et que je veux utiliser la méthode `head`.

Cela fonctionnerait aussi avec le code suivant :

```
<?php
app('panel')->head()
```

Ici, je cherche directement dans le conteneur une instance de la classe. Je pourrais aussi l'écrire comme suit :

```
<?php
App::make('panel')->head()
```

Cette fois, j'utilise la façade de l'application (donc du conteneur parce que l'application en est une extension) et je demande de créer une instance de la classe référencée par `'panel'`.

Il ne reste plus qu'à informer Laravel que notre fournisseur existe. Cela se fait dans `config/app.php`. Pour installer le package, on a écrit ce qui suit :

```
<?php
Collective\Html\HtmlServiceProvider::class,
```

On change cette ligne pour pointer sur notre fournisseur :

Copyright © 2016 Eyrolles.

```
<?php
App\Services\Html\HtmlServiceProvider::class,
```

Il ne nous reste plus qu'à renseigner la façade pour le panel :

```
<?php
'Panel'=>App\Services\Html\PanelFacade::class,
```

Tout devrait bien fonctionner !

Si vous rencontrez des difficultés, surtout si vous avez une erreur sur le fait qu'une classe n'existe pas, exécutez les commandes suivantes :

```
composer dumpautoload
php artisan clear-compiled
```

En résumé

- Le conteneur de dépendances est la clé du fonctionnement de Laravel.
- On enregistre un composant avec un fournisseur de services.
- On simplifie la syntaxe d'accès à une classe avec une façade.

Copyright © 2016 Eyrolles.

Copyright © 2016 Eyro les.

21 La localisation

Lorsqu'on crée un site, on veut souvent le proposer dans plusieurs langues. On parle alors d'internationalisation (`i18n`) et de localisation (`L10n`). L'internationalisation consiste à préparer une application pour l'adapter à différents contextes linguistiques. La localisation consiste quant à elle à ajouter un composant spécifique à une langue.

C'est un sujet assez complexe qui ne se limite pas à la traduction des textes, mais qui affecte aussi la représentation des dates, la gestion des pluriels, parfois la mise en page…

Dans ce chapitre, nous allons découvrir ce que nous propose Laravel dans ce domaine et nous rendrons notre petit blog disponible en français et en anglais.

On utilise les abréviations `i18n`, parce qu'il y a 18 lettres entre le `i` et le `n` de *internationalisation*, et `L10n` parce qu'il y a 10 lettres entre le `L` et le `n` de *Localisation*.

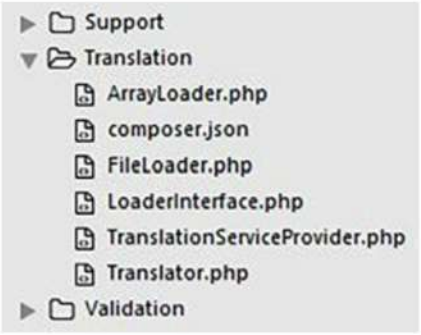
Le principe

Façade Lang et classe Translator

Laravel est équipé de la façade `Lang` pour générer des chaînes de caractères dans plusieurs langues.

Vous savez qu'une façade cache une classe. Toutes les classes qui sont concernées par la traduction se trouvent dans le dossier `Illuminate\Translation`.

Copyright © 2016 Eyrolles.



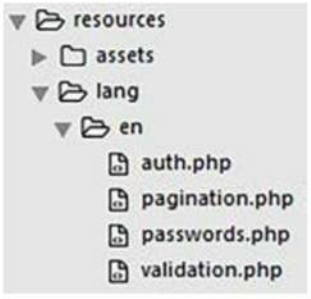
Le dossier pour les traductions

On y trouve très logiquement :

- un fournisseur pour réaliser toutes les initialisations ;
- un fichier `composer.json` pour les dépendances ;
- une classe `FileLoader` avec son interface `LoaderInterface` pour gérer les fichiers de traduction ;
- une classe `ArrayLoader` pour charger les messages ;
- et enfin la classe principale `Translator` qui est mise en action par la façade.

Fichiers de langue

On a déjà parlé des fichiers de langue. Lorsque Laravel est installé, on a l'architecture suivante.



Les dossiers de langue

Il n'est prévu au départ que l'anglais (`en`) avec quatre fichiers. Pour ajouter une langue, il suffit de créer un nouveau dossier, par exemple `fr` pour le français et il faut évidemment avoir le même contenu, comme nous allons le voir.

Vous n'allez pas être obligés de tout traduire vous-mêmes ! Certains s'en sont déjà chargés avec le package *https://github.com/caouecs/Laravel-lang*. Nous l'avons déjà utilisé ici. Téléchargez-le et copiez le dossier du français si vous ne l'avez pas encore ou si vous l'avez supprimé.

Copyright © 2016 Eyrolles.

Le dossier du français

Vous disposez alors de deux versions linguistiques des messages de base de Laravel.

Voyons comment sont constitués ces fichiers. Prenons le plus léger (`pagination.php`) :

```
<?php
return [
    'previous'=>'&laquo; Previous',
    'next'=>'Next &raquo;',
];
```

On se contente de renvoyer un tableau avec des clés et des valeurs. On ne pourrait pas faire plus simple ! Voici le même fichier en version française :

```
<?php
return [
    'previous'=>'&laquo; Précédent',
    'next'=>'Suivant &raquo;',
];
```

On retrouve évidemment les mêmes clés avec des valeurs adaptées à la langue.

Fonctionnement

On dispose de quelques méthodes pour tester et récupérer ces valeurs.

Avec le code suivant, j'obtiens « `Previous` », donc la version anglaise :

```
<?php
Lang::get('pagination.previous');
```

Copyright © 2016 Eyrolles.

Comment faire pour obtenir la version française ? Regardez dans le fichier `config/app.php` :

```
<?php
'locale'=>'en',
```

C'est ici qu'est fixée la langue en cours. Changez la valeur :

```
<?php
'locale'=>'fr',
```

Avec le même code, vous allez obtenir « `Précédent` ».

Évidemment, il est possible de changer cette valeur dans le code avec la méthode `setLocale` :

```
<?php
App::setLocale('fr');
```

Comme vous aurez de multiples endroits dans le code où vous allez récupérer des valeurs, il existe un *helper* :

```
<?php
trans('pagination.previous');
```

On peut aussi vérifier qu'une clé existe avec la méthode `has` :

```
<?php
Lang::has('pagination.previous');
```

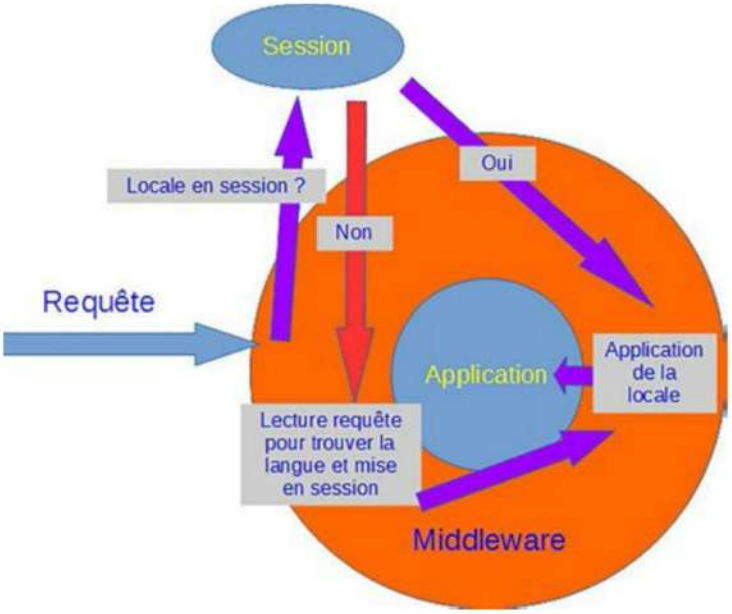
Ce sont les méthodes fondamentales qui seront suffisantes pour le présent chapitre. Vous trouverez des compléments d'information dans la documentation : *http://laravel.com/docs/5.2/localization*.

Le middleware

Quand un utilisateur choisit une langue, il faut se la rappeler. On utilisera donc la session pour mémoriser son choix. Lorsqu'un utilisateur arrive sans avoir encore choisi, il faut lui présenter une langue. Pour cela, il est possible d'aller chercher la langue utilisée au niveau de la requête.

Maintenant, où allons-nous placer le code pour gérer tout cela ? C'est un *middleware* qui va logiquement s'en occuper.

Copyright © 2016 Eyrolles.

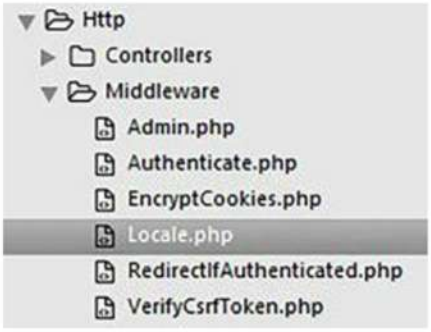


Gestion de la locale

Lorsque la requête arrive, elle est prise en charge par le *middleware*. Là, on regarde si l'utilisateur a une session active avec une localisation. Si c'est le cas, on applique cette locale et on envoie la requête à l'étape suivante. Si ce n'est pas le cas, on regarde dans l'en-tête de la requête la langue envoyée par le navigateur, on la mémorise dans la session et on l'applique. On envoie alors la requête à l'étape suivante.

Nous allons créer ce *middleware* avec `Artisan` :

```
php artisan make:middleware Locale
```



Le nouveau middleware

Je l'ai nommé `Locale` pour représenter sa fonction. Voici le code généré par défaut :

Copyright © 2016 Eyrolles.

```
<?php
namespace App\Http\Middleware;

use Closure;

class Locale
{
  /**
   * Handle an incoming request.
   *
   * @param \Illuminate\Http\Request $request
   * @param \Closure $next
   * @return mixed
   */
  public function handle($request, Closure $next)
  {
    return $next($request);
  }
}
```

Ajoutons notre code :

```
<?php
namespace App\Http\Middleware;

use Closure;

class Locale
{
  protected $languages=['en','fr'];

  public function handle($request, Closure $next)
  {
    if (!session()->has('locale'))
    {
      session()->put('locale', $request->getPreferredLanguage($this-
>languages));
    }
    app()->setLocale(session('locale'));
    return $next($request);
  }
}
```

Le code reprend strictement le schéma précédent.

Il ne nous reste plus qu'à dire à Laravel de prendre en compte ce *middleware* dans `app/Http/Kernel.php` :

```
<?php
protected $middlewareGroups=[
  'web'=>[
    ...
    \App\Http\Middleware\Locale::class,
  ],
  ...
];
```


Copyright © 2016 Eyrolles.

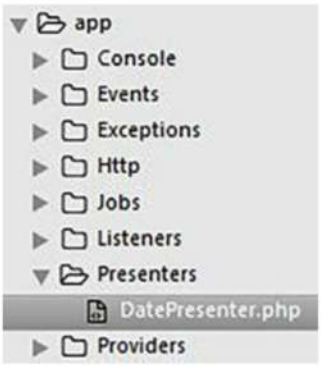
Les dates

Il ne vous a sans doute pas échappé que les dates ne sont pas présentées de la même manière selon les langues utilisées. En France, nous utilisons le format `jj/mm/aaaa` alors que les Américains utilisent le format `mm/jj/aaaa`. Il faut donc que les dates de création des articles apparaissent au bon format dans chaque cas.

Comme ce formatage doit être appliqué pour toutes les dates, la façon la plus simple et élégante de procéder est de créer un `accessor` sur la propriété. Ainsi, chaque fois qu'on extraira une date à partir du modèle, on la formatera au passage. On pourrait écrire ceci directement dans le modèle `Post`. Cependant, pour généraliser, imaginons qu'on peut en avoir besoin pour d'autres modèles.

Dans ce type de situation, la création d'un `trait` est pertinente pour introduire une fonctionnalité sans passer par l'héritage. On utilisera une version simplifiée du *design pattern* `Décorateur` (`Decorator`). Dans Laravel, on appelle cela un `Model Presenter`.

Définissons un dossier pour les `Presenter` et créons celui dont nous avons besoin.



Le Presenter pour les dates

Voici son code :

```
<?php
namespace App\Presenters;

use Carbon\Carbon;

trait DatePresenter
{
  public function getCreatedAtAttribute($date)
  {
    return $this->getDateFormated($date);
  }

  public function getUpdatedAtAttribute($date)
  {
```

Copyright © 2016 Eyrolles.

```
    return $this->getDateFormated($date);
  }

  private function getDateFormated($date)
  {
    return Carbon::parse($date)->format(config('app.locale')=='fr' ? 'd/m/
Y':'m/d/Y');
  }
```

On voit que, selon la valeur de la locale, on va appliquer le formatage adapté de la date extraite.

Il ne nous reste plus qu'à utiliser ce `trait` dans le modèle `Post` :

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;
use App\Presenters\DatePresenter;

class Post extends Model
{
  use DatePresenter;

  protected $fillable=['titre','contenu','user_id'];

  public function user()
  {
    return $this->belongsTo('App\User');
  }

  public function tags()
  {
    return $this->belongsToMany('App\Tag');
  }
}
```

Route et contrôleur

L'utilisateur doit pouvoir changer la langue s'il le désire. Il faut donc ajouter une route :

```
<?php
Route::get('language', 'PostController@language');
```

Et voici le contrôleur modifié :

```
<?php
namespace App\Http\Controllers;
```

Copyright © 2016 Eyrolles.

```
use App\Repositories\PostRepository;
use App\Repositories\TagRepository;
use App\Http\Requests\PostRequest;

class PostController extends Controller
{
  protected $postRepository;
  protected $nbrPerPage=4;

  public function __construct(PostRepository $postRepository)
  {
    $this->middleware('auth', ['except'=>['index', 'indexTag',
'language']]);
    $this->middleware('admin', ['only'=>'destroy']);
    $this->postRepository=$postRepository;
  }

 public function index()
  {
    $posts=$this->postRepository->getWithUserAndTagsPaginate($this-
>nbrPerPage);
    $links=$posts->render();
    return view('posts.liste', compact('posts', 'links'));
  }

  public function create()
  {
    return view('posts.add');
  }

  public function store(PostRequest $request, TagRepository $tagRepository)
  {
    $inputs=array_merge($request->all(), ['user_id'=>$request->user()->id]);
    $post=$this->postRepository->store($inputs);
    if (isset($inputs['tags']))
    {
      $tagRepository->store($post, $inputs['tags']);
    }
    return redirect(route('post.index'));
  }

  public function destroy($id)
  {
    $this->postRepository->destroy($id);
    return redirect()->back();
  }

 public function indexTag($tag)
  {
    $posts=$this->postRepository->getWithUserAndTagsForTagPaginate($tag,
$this->nbrPerPage);
    $links=$posts->render();
    return view('posts.liste', compact('posts', 'links'))
    ->with('info', trans('blog.search') . $tag);
  }
```

Copyright © 2016 Eyrolles.

```
  public function language()
  {
    session()->set('locale', session('locale')=='fr' ? 'en':'fr');
    return redirect()->back();
  }
}
```

On voit une nouvelle fonction `language`. De plus, la fonction `indexTag` a été modifiée pour tenir compte des langues. Donc, on va pouvoir passer d'une langue à l'autre avec l'URL *.../language*.

La réalisation de la localisation

Il nous faut aussi créer les fichiers de localisation.

Les fichiers de localisation

Voici le contenu pour `resources/lang/en/blog.php` :

```
<?php
return [
  'site'=>'My nice site',
  'title'=>'My nice blog',
  'creation'=>'Create an article',
  'login'=>'Login',
  'logout'=>'Logout',
  'delete'=>'Delete this post',
  'confirm'=>'Really delete this post ?',
  'on'=>'on',
  'search'=>'Results for tag : '
];
```

Copyright © 2016 Eyrolles.

Voici celui pour `resources/lang/fr/blog.php` :

```
<?php
return [
  'site'=>'Mon joli site',
  'title'=>'Mon joli blog',
  'creation'=>'Créer un article',
  'login'=>'Se connecter',
  'logout'=>'Déconnexion',
  'delete'=>'Supprimer cet article',
  'confirm'=>'Vraiment supprimer cet article ?',
  'on'=>'le',
  'search'=>'Résultats pour la recherche du mot-clé : '
];
```

Les vues

Il ne nous reste plus qu'à modifier les vues pour que le tout fonctionne. Voici d'abord le template (`views/template/php`) :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>{{ trans('blog.site') }}</title>
    {!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/
bootstrap.min.css') !!}
    {!! Html::style('https://netdna.bootstrapcdn.com/bootstrap/3.3.6/css/
bootstrap-theme.min.css') !!}
    <!--[if lt IE 9]>
      {{ Html::style('https://oss.maxcdn.com/libs/html5shiv/3.7.2/html5shiv.
js') }}
      {{ Html::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.
min.js') }}
    <![endif]-->
    <style>textarea {resize:none;}</style>
  </head>
  <body>
    <header class="jumbotron">
      <div class="container">
        <h1 class="page-header">{!! link_to_route('post.index', trans('blog.
title')) !!}</h1>
        @yield('header')
      </div>
    </header>
    <div class="container">
      @yield('contenu')
    </div>
  </body>
</html>
```

Copyright © 2016 Eyrolles.

Il est modifié pour la traduction du titre du site dans l'en-tête :

```
<title>{{ trans('blog.site') }}</title>
```

et le titre/lien sur la page :

```
<h1 class="page-header">{!! link_to_route('post.index', trans('blog.
title')) !!}</h1>
```

Voici le blog transformé (`views/posts/liste.blade.php`) :

```
@extends('template')

@section('header')
  <div class="btn-group pull-right">
    {!! link_to('language', session('locale')=='fr' ? 'English':'Français',
['class'=>'btn btn-primary']) !!}
    @if(Auth::check())
      {!! link_to_route('post.create', trans('blog.creation'), [],
['class'=>'btn btn-info']) !!}
      {!! link_to('logout', trans('blog.logout'), ['class'=>'btn btn-
warning']) !!}
    @else
      {!! link_to('login', trans('blog.login'), ['class'=>'btn btn-info
pull-right']) !!}
    @endif
  </div>
@endsection
@section('contenu')
  @if(isset($info))
    <div class="row alert alert-info">{{$info}}</div>
  @endif
  {!! $links !!}
  @foreach($posts as $post)
    <article class="row bg-primary">
      <div class="col-md-12">
        <header>
          <h1>{{$post->titre}}
            <div class="pull-right">
              @foreach($post->tags as $tag)
                {!! link_to('post/tag/' . $tag->tag_url, $tag->tag,
['class'=>'btn btn-xs btn-info']) !!}
              @endforeach
            </div>
         </h1>
        </header>
        <hr>
       <section>
          <p>{{$post->contenu}}</p>
         @if (Auth::check() and Auth::user()->admin)
           {!! Form::open(['method'=>'DELETE', 'route'=>['post.destroy',
$post->id]]) !!}
```

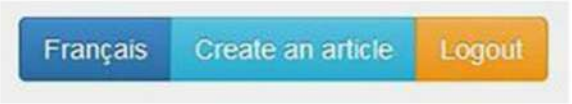
Copyright © 2016 Eyrolles.

```
                {!! Form::submit(trans('blog.delete'), ['class'=>'btn btn-
danger btn-xs', 'onclick'=>'return confirm(\'' . trans('blog.confirm')
. '\')']) !!}
              {!! Form::close() !!}
            @endif
            <em class="pull-right">
              <span class="glyphicon glyphicon-pencil"></span>{{ $post->user-
>name . ' ' . trans('blog.on') . ' ' . $post->created_at }}
            </em>
          </section>
        </div>
      </article>
      <br>
    @endforeach
    {!! $links !!}
@endsection
```

Les modifications portent sur tous les textes. Je vous laisse analyser le code. En français, rien ne change par rapport à ce que nous avions. Passons en anglais en cliquant sur le bouton ***English***.

Le bouton de changement de langue

- Les boutons changent d'aspect.



Les boutons en anglais

- Le titre de la page est en anglais.

Le titre de la page en anglais

- Le titre/lien est en anglais.

Le titre en anglais

Copyright © 2016 Eyrolles.

• Les dates sont formatées correctement.

Date formatée

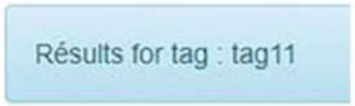
• Le bouton de suppression est aussi en anglais.

Le bouton de suppression

• Il en va de même pour le message d'alerte.

Le message d'alerte en anglais

• Le message de la recherche par par mot-clé est aussi adapté.



Le message de la recherche par mot-clé

Il est parfois nécessaire de vider le dossier `storage/framework/views` pour purger le cache des vues. Toutefois ne supprimez pas le fichier `.gitignore`.

Les noms des contrôles

Il faudrait aussi traiter la vue de création d'un article, mais c'est exactement le même principe. Il y a toutefois un élément à prendre en compte, c'est le nom des contrôles de saisie. Vous vous rappelez qu'on les a appelés `titre`, `contenu` et `tags`. Ces noms ne sont pas visibles tant qu'il n'y a pas de souci de validation ; ils sont transmis dans

Copyright © 2016 Eyrolles.

le texte de l'erreur. Nous devons éviter d'obtenir en langue anglaise le message « `The titre field is required.` ».

Si vous regardez dans le fichier `resources/lang/fr/validation.php`, vous trouvez le tableau suivant :

```
<?php
'attributes'=>[
  "name"=>"Nom",
  "username"=>"Pseudo",
  ...
],
```

Ici, on fait correspondre le nom d'un attribut avec sa version linguistique pour justement avoir quelque chose de cohérent au niveau du message d'erreur. Donc, dans notre situation, étant donné que nous avons francisé le nom des contrôles, il faut ajouter la version anglaise dans le fichier `resources/lang/en/validation.php` :

```
<?php
'attributes'=>[
  "titre"=>"Title",
  "contenu"=>"Content",
],
```

Évidemment, si on avait prévu des noms anglais, il aurait fallu définir à l'inverse des appellations françaises dans le fichier `resources/lang/fr/validation.php`.

Le but de ce chapitre est de montrer comment localiser l'interface d'une page web. S'il s'agit d'adapter le contenu selon la langue, il faut intervenir au niveau de l'URL. En effet un principe fondamental du Web veut que, pour une certaine URL, on renvoie toujours le même contenu.

En résumé

- Laravel possède les outils de base pour la localisation.
- Il faut créer autant de fichiers de localisation qu'on a de langues à traiter.
- La localisation doit s'effectuer au niveau d'un *middleware*.
- Le formatage des dates peut s'effectuer facilement avec un `Presenter`.
- Il faut prévoir la version linguistique des attributs.

Copyright © 2016 Eyrolles.

Copyright © 2016 Eyro les.

22 Ajax

Ajax est une technologie JavaScript fort répandue qui permet d'envoyer des requêtes au serveur et de recevoir des réponses sans rechargement de la page. Il est par ce moyen possible de modifier dynamiquement le DOM, donc une partie de la page.

Dans ce chapitre, nous allons expliquer comment mettre en œuvre Ajax avec Laravel. Nous partirons d'une nouvelle installation de Laravel et mettrons en place l'authentification prévue avec `Artisan`. Nous la modifierons ensuite afin d'utiliser Ajax pour l'enregistrement d'un utilisateur. Pour ajouter de l'intérêt à l'exemple, nous placerons le formulaire dans une page modale.

Créez une nouvelle installation, puis utilisez la commande `Artisan` suivante :

```
php artisan make:auth
```

Créez aussi une base de données et effectuez la migration.

Vérifiez que tout fonctionne correctement. On va un peu changer le code…

Les vues

Template

Dans le template par défaut (`resources/views/layouts/app.blade.php`), on apporte deux modifications. Voici le code résultant :

```
<!DOCTYPE html>
<html lang="en">
<head>
```

Copyright © 2016 Eyrolles.

```
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Laravel</title>
  <!-- Fonts -->
  <link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.4.0/css/
font-awesome.min.css" rel='stylesheet' type='text/css'>
  <link href="https://fonts.googleapis.com/css?family=Lato:100,300,400,700"
rel='stylesheet' type='text/css'>
  <!-- Styles -->
  {{-- <link href="{{ elixir('css/app.css') }}" rel="stylesheet"> --}}
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.
min.css" rel="stylesheet">
  <style>
    body {
      font-family: 'Lato';
    }
    .fa-btn {
      margin-right: 6px;
    }
  </style>
</head>
<body id="app-layout">
  <nav class="navbar navbar-default">
    <div class="container">
      <div class="navbar-header">
        <!-- Collapsed Hamburger -->
        <button type="button" class="navbar-toggle collapsed" data-
toggle="collapse" data-target="#spark-navbar-collapse">
          <span class="sr-only">Toggle Navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <!-- Branding Image -->
        <a class="navbar-brand" href="{{ url('/') }}">
          Laravel
        </a>
      </div>

     <div class="collapse navbar-collapse" id="spark-navbar-collapse">
        <!-- Left Side Of Navbar -->
        <ul class="nav navbar-nav">
          <li><a href="{{ url('/') }}">Home</a></li>
        </ul>
        <!-- Right Side Of Navbar -->
        <ul class="nav navbar-nav navbar-right">
          <!-- Authentication Links -->
          @if (Auth::guest())
            <li><a href="{{ url('/login') }}">Login</a></li>
          @else
            <li class="dropdown">
              <a href="#" class="dropdown-toggle" data-toggle="dropdown"
role="button" aria-expanded="false">
                {{ Auth::user()->name }}
                 <span class="caret"></span>
              </a>
```

Copyright © 2016 Eyrolles.

```
                <ul class="dropdown-menu" role="menu">
                  <li><a href="{{ url('/logout') }}"><i class="fa fa-btn fa-
sign-out"></i>Logout</a></li>
                </ul>
              </li>
            @endif
          </ul>
        </div>
      </div>
    </nav>

    @yield('content')
    <!-- JavaScripts -->
    {{-- <script src="{{ elixir('js/app.js') }}"></script> --}}
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.4/jquery.
min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.
min.js"></script>
    @yield('scripts')
</body>
</html>
```

Les deux modifications sont :

- la suppression du code pour l'option `Register` dans le menu ; on ne garde plus que `Login` :

```
@if (Auth::guest())
    <li><a href="{{ url('/login') }}">Login</a></li>
@else
```

- une nouvelle section pour insérer un script :

```
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.
min.js"></script>
@yield('scripts')
```

La barre de menus présente donc l'aspect suivant lorsque personne n'est authentifié.

La barre de menu sans Register

Copyright © 2016 Eyrolles.

Vue login

C'est cette vue (*resources/views/auth/login.blade.php*) qui subira le plus de changements. En voici le code :

```
@extends('layouts.app')

@section('content')
<div class="container">
  <div class="row">
    <div class="col-md-8 col-md-offset-2">
      <div class="alert alert-success alert-dismissible hidden">
        You are now registered, you can login.
      </div>

     <div class="panel panel-default">
        <div class="panel-heading">Login</div>
        <div class="panel-body">
          <form class="form-horizontal" role="form" method="POST"
action="{{ url('/login') }}">
            {!! csrf_field() !!}
            <div class="form-group{{ $errors->has('email') ? ' has-
error' : '' }}">
              <label class="col-md-4 control-label">E-Mail Address</label>
              <div class="col-md-6">
                <input type="email" class="form-control" name="email"
value="{{ old('email') }}">
                @if ($errors->has('email'))
                  <span class="help-block">
                    <strong>{{ $errors->first('email') }}</strong>
                  </span>
                @endif
              </div>
            </div>

            <div class="form-group{{ $errors->has('password') ? ' has-
error' : '' }}">
              <label class="col-md-4 control-label">Password</label>
              <div class="col-md-6">
                <input type="password" class="form-control" name="password">
                @if ($errors->has('password'))
                  <span class="help-block">
                    <strong>{{ $errors->first('password') }}</strong>
                  </span>
                @endif
              </div>
            </div>

            <div class="form-group">
              <div class="col-md-6 col-md-offset-4">
                <div class="checkbox">
                  <label>
                    <input type="checkbox" name="remember">Remember Me
                  </label>
                </div>
              </div>
```

Copyright © 2016 Eyrolles.

```
            </div>

            <div class="form-group">
              <div class="col-md-6 col-md-offset-4">
                <button type="submit" class="btn btn-primary">
                  <i class="fa fa-btn fa-sign-in"></i>Login
                </button>
                <a class="btn btn-link" id="register" href="#">Register</a>
                <a class="btn btn-link" href="{{ url('/password/
reset') }}">Forgot Your Password?</a>
              </div>
            </div>
          </form>
        </div>
      </div>
    </div>
  </div>
</div>

<!-- Modal -->
<div class="modal fade" id="myModal" tabindex="-1" role="dialog" aria-
labelledby="myModalLabel" aria-hidden="true">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <button type="button" class="close" data-dismiss="modal" aria-
label="Close"><span aria-hidden="true">×</span></button>
        <h4 class="modal-title" id="myModalLabel">Register</h4>
      </div>
      <div class="modal-body">
        <form id="formRegister" class="form-horizontal" role="form"
method="POST" action="{{ url('/register') }}">
          {!! csrf_field() !!}
          <div class="form-group">
            <label class="col-md-4 control-label">Name</label>
            <div class="col-md-6">
              <input type="text" class="form-control" name="name">
              <small class="help-block"></small>
            </div>
          </div>

          <div class="form-group">
            <label class="col-md-4 control-label">E-Mail Address</label>
            <div class="col-md-6">
              <input type="email" class="form-control" name="email">
              <small class="help-block"></small>
            </div>
          </div>

          <div class="form-group">
            <label class="col-md-4 control-label">Password</label>
            <div class="col-md-6">
              <input type="password" class="form-control" name="password">
              <small class="help-block"></small>
            </div>
          </div>
```

Copyright © 2016 Eyrolles.

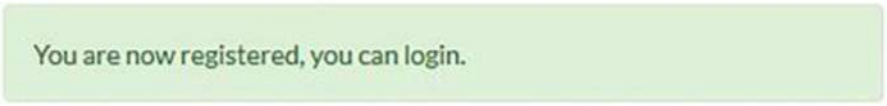
```
            <div class="form-group">
              <label class="col-md-4 control-label">Confirm Password</label>
              <div class="col-md-6">
                <input type="password" class="form-control" name="password_
confirmation">
              </div>
            </div>

            <div class="form-group">
              <div class="col-md-6 col-md-offset-4">
                <button type="submit" class="btn btn-primary">Register
                </button>
              </div>
            </div>
          </form>
        </div>
      </div>
    </div>
</div>

@endsection

@section('scripts')
<script>
$(function() {
  $('#register').click(function() {
    $('#myModal').modal();
  });

  $(document).on('submit', '#formRegister', function(e) {
    e.preventDefault();
    $('input+small').text('');
    $('input').parent().removeClass('has-error');
    $.ajax({
      method: $(this).attr('method'),
      url: $(this).attr('action'),
      data: $(this).serialize(),
      dataType: "json"
    })
    .done(function(data) {
      $('.alert-success').removeClass('hidden');
      $('#myModal').modal('hide');
    })
    .fail(function(data) {
      $.each(data.responseJSON, function (key, value) {
        var input='#formRegister input[name='+key+']';
        $(input+'+small').text(value);
        $(input).parent().addClass('has-error');
      });
    });
  });
})
</script>
@endsection
```

Les changements sont les suivants :

Copyright © 2016 Eyrolles.

• ajout d'une barre d'information cachée par défaut (`hidden`) :

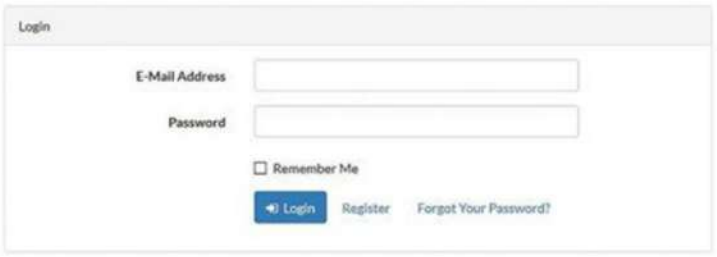
```
<div class="alert alert-success alert-dismissible hidden">
    You are now registered, you can login.
</div>
```



La barre d'information

• ajout d'un lien pour l'enregistrement :

```
<a class="btn btn-link" id="register" href="#">Register</a>
<a class="btn btn-link" href="{{ url('/password/reset') }}">Forgot Your
Password?</a>
```



Le lien pour s'enregistrer

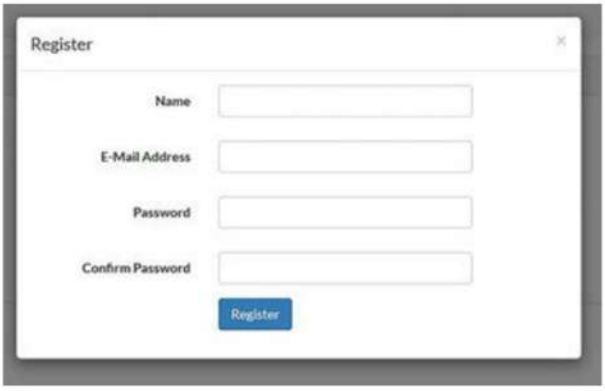
• ajout du code de la page modale qui héberge le formulaire :

```
<div class="modal fade" id="myModal" tabindex="-1" role="dialog" aria-
labelledby="myModalLabel" aria-hidden="true">
  ...
</div>
```

C'est du codage classique avec `Bootstrap`. Remarquez pour chaque contrôle du formulaire la partie qui servira pour l'affichage éventuel des erreurs de validation :

```
<small class="help-block"></small>
```

Copyright © 2016 Eyrolles.



La fenêtre modale d'enregistrement

- ajout du code JavaScript pour la gestion du formulaire d'enregistrement :

```
$(function() {
  $('#register').click(function() {
    $('#myModal').modal();
  });
  $(document).on('submit', '#formRegister', function(e) {
    ...
  });
})
```

JavaScript

Étudions de plus près la partie JavaScript.

À la soumission du formulaire, il faut éviter l'envoi normal :

```
e.preventDefault();
```

Puis on supprime toute trace d'erreur précédente :

```
$('input+small').text('');
$('input').parent().removeClass('has-error');
```

Il faut ensuite envoyer la requête avec les valeurs saisies en se simplifiant la vie avec une sérialisation :

```
$.ajax({
  method: $(this).attr('method'),
  url: $(this).attr('action'),
  data: $(this).serialize(),
  dataType: "json"
})
```

Copyright © 2016 Eyrolles.

Si la validation est correcte, il faut afficher la barre d'information et cacher la fenêtre modale :

```
.done(function(data) {
  $('.alert-success').removeClass('hidden');
  $('#myModal').modal('hide');
})
```

En cas d'erreur dans la validation, on balaye les erreurs et on met en place le style :

```
.fail(function(data) {
  $.each(data.responseJSON, function (key, value) {
    var input='#formRegister input[name='+key+']';
    $(input+'+small').text(value);
    $(input).parent().addClass('has-error');
  });
});
```

Par exemple, voici ce qu'on obtient si on ne remplit par le contrôle du nom :

Nom non renseigné

Le traitement

Contrôleur

Voici le nouveau contrôleur `app/Http/Controllers/Auth/AuthController` :

```
<?php
namespace App\Http\Controllers\Auth;

use App\User;
use Validator;
use App\Http\Controllers\Controller;
```

Copyright © 2016 Eyrolles.

```
use Illuminate\Http\Request;
use Illuminate\Foundation\Auth\ThrottlesLogins;
use Illuminate\Foundation\Auth\AuthenticatesAndRegistersUsers;

class AuthController extends Controller
{
  /* |--------------------------------------------------------------
  | Registration & Login Controller
  |--------------------------------------------------------------
  |
  | This controller handles the registration of new users, as well as the
  | authentication of existing users. By default, this controller uses
  | a simple trait to add these behaviors. Why don't you explore it?
  |
  */

  use AuthenticatesAndRegistersUsers, ThrottlesLogins;

 /**
   * Where to redirect users after login / registration.
   *
   * @var string
   */
  protected $redirectTo='/home';

  /**
   * Create a new authentication controller instance.
   *
   * @return void
   */
  public function __construct()
  {
    $this->middleware('guest', ['except'=>'logout']);
    $this->middleware('ajax', ['only'=>'register']);
  }

 /**
   * Get a validator for an incoming registration request.
   *
   * @param array $data
   * @return \Illuminate\Contracts\Validation\Validator
   */
  protected function validator(array $data)
  {
    return Validator::make($data, [
      'name'=>'required|max:255',
      'email'=>'required|email|max:255|unique:users',
      'password'=>'required|confirmed|min:6',
    ]);
  }

  /**
   * Create a new user instance after a valid registration.
   *
   * @param array $data
  * @return User
   */
```

Copyright © 2016 Eyrolles.

```
  protected function create(array $data)
  {
    return User::create([
      'name'=>$data['name'],
      'email'=>$data['email'],
      'password'=>bcrypt($data['password']),
    ]);
  }

  /**
   * Handle a registration request for the application.
   *
   * @param \Illuminate\Http\Request $request
   * @return \Illuminate\Http\Response
   */
  public function register(Request $request)
  {
    $validator=$this->validator($request->all());
    if ($validator->fails()) {
      $this->throwValidationException(
        $request, $validator
      );
    }
    $this->create($request->all());
    return response()->json();
  }
}
```

La méthode `register` se situe normalement dans le `trait`. Cependant, comme nous avons besoin d'en modifier le code, nous surchargeons cette méthode dans le contrôleur. Les seules différences avec le code du `trait` sont :

- pas d'authentification immédiate à l'enregistrement ;
- réponse JSON.

Middleware

On peut aussi remarquer dans le contrôleur la présence d'un nouveau *middleware* `ajax` :

```
<?php
$this->middleware('ajax', ['only'=>'register']);
```

En effet, on veut que notre méthode `register` ne soit accessible que si on reçoit une requête `Ajax`. Ce *middleware* n'existe pas par défaut dans Laravel ; il faut donc le créer (`app/Http/Middlewares/Ajax.php`) :

Copyright © 2016 Eyrolles.

```
<?php
namespace App\Http\Middleware;

use Closure;

class Ajax
{
  public function handle($request, Closure $next)
  {
    if ($request->ajax())
    {
      return $next($request);
    }
    abort(404);
  }
}
```

La méthode `ajax` de la requête nous indique s'il s'agit d'une requête de ce type ; si ce n'est pas le cas, on renvoie une erreur 404.

Il faut informer Laravel que notre *middleware* existe (`app/Http/Kernel.php`) :

```
<?php
protected $routeMiddleware=[
  'auth'=> \App\Http\Middleware\Authenticate::class,
  'auth.basic'=> \Illuminate\Auth\Middleware\
AuthenticateWithBasicAuth::class,
  'guest'=> \App\Http\Middleware\RedirectIfAuthenticated::class,
  'throttle'=> \Illuminate\Routing\Middleware\ThrottleRequests::class,
  'ajax'=> \App\Http\Middleware\Ajax::class,
];
```

En résumé

- Ajax est facile à mettre en œuvre avec Laravel.
- La sérialisation permet de mettre en forme les entrées du formulaire.
- On peut prévoir un *middleware* particulier pour les requêtes Ajax.

Copyright © 2016 Eyrolles.

23 Les tests unitaires

Les développeurs PHP n'ont pas été habitués à réaliser des tests pour leurs applications. Cela est dû à l'histoire de ce langage, qui n'était au départ qu'une possibilité de scripter au milieu du code HTML, mais qui s'est peu à peu développé comme un langage de plus en plus évolué. Les créateurs de frameworks ont initié une autre façon d'organiser le code de PHP ; en particulier, ils ont mis en avant la séparation des tâches qui a rendu possible la création de tests.

Laravel a été pensé pour intégrer des tests. Il comporte une infrastructure élémentaire et des *helpers*. Ce que je vais vous présenter ici est une simple introduction à ce domaine qui mériterait à lui seul un ouvrage spécifique. Je m'efforcerai de vous démontrer l'utilité de créer des tests, comment les préparer et comment les isoler.

Lorsqu'on développe avec PHP, on effectue forcément des tests, au moins manuels. Par exemple, si vous créez un formulaire, vous l'utilisez, entrez diverses informations, essayez de fausses manœuvres… Imaginez que tout cela soit automatisé et que vous n'ayez qu'à cliquer pour lancer tous les tests. C'est le propos de ce chapitre.

Vous pouvez aussi vous dire qu'écrire des tests conduit à du travail supplémentaire, que ce n'est pas toujours facile, que ce n'est pas nécessaire dans tous les cas. C'est à vous de voir si vous avez besoin d'en créer ou pas. Pour de petites applications, la question reste ouverte. En revanche, dès qu'une application prend de l'ampleur ou lorsqu'elle est conduite par plusieurs personnes, il devient vite nécessaire de créer des tests automatisés.

Copyright © 2016 Eyrolles.

L'intendance des tests

PHPUnit

Laravel utilise PHPUnit (*http://phpunit.de/*) pour effectuer les tests. C'est un framework créé par Sebastian Bergmann qui fonctionne à partir d'assertions.

Ce framework est installé comme dépendance de Laravel en mode développement :

```
"require-dev": {
  "fzaninotto/faker": "~1.4",
  "mockery/mockery": "0.9.*",
  "phpunit/phpunit": "~4.0",
  "symfony/css-selector": "2.8.*|3.0.*",
  "symfony/dom-crawler": "2.8.*|3.0.*"
},
```

Vous pouvez aussi utiliser le fichier `phar` (*http://phpunit.de/#download*) en le plaçant à la racine de votre application. Vous êtes prêts à tester !

La version 5 de PHPUnit nécessite PHP 5.6.

Vérifiez que cela fonctionne en entrant la commande suivante :

```
php phpunit.phar -h
```

Vous obtenez alors la liste de toutes les commandes disponibles.

Si vous utilisez la version installée avec Laravel, la commande est la suivante :

```
php vendor\phpunit\phpunit\phpunit -h
```

Je vous conseille de créer un alias.

Dans tous les exemples de ce chapitre, j'utiliserai le fichier `phar`.

Intendance de Laravel

Un des dossiers de Laravel est consacré aux tests.

Les dossier des tests

Copyright © 2016 Eyrolles.

Vous avez déjà deux fichiers. Voilà `TestCase.php` :

```
<?php
class TestCase extends Illuminate\Foundation\Testing\TestCase
{
  /**
   * The base URL to use while testing the application.
   *
   * @var string
   */
  protected $baseUrl='http://localhost';

  /**
   * Creates the application.
   *
   * @return \Illuminate\Foundation\Application
   */
  public function createApplication()
  {
    $app=require __DIR__.'/../bootstrap/app.php';
    $app->make(Illuminate\Contracts\Console\Kernel::class)->bootstrap();
    return $app;
  }
}
```

Cette classe est chargée de créer une application pour les tests dans un environnement spécifique (ce qui permet de mettre en place une configuration adaptée aux tests).

Elle étend la classe `Illuminate\Foundation\Testing\TestCase`, qui elle-même étend la classe `PHPUnit_Framework_TestCase` en lui ajoutant quelques fonctionnalités bien pratiques.

Toutes les classes de test que vous allez créer devront étendre cette classe `TestCase`. Il y a un exemple de test déjà présent `ExampleTest.php` :

```
<?php
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
  /**
   * A basic functional test example.
   *
   * @return void
   */
  public function testBasicExample()
  {
    $this->visit('/')
    ->see('Laravel 5');
  }
}
```

Copyright © 2016 Eyrolles.

Sans entrer pour le moment dans le code, sachez simplement qu'on envoie une requête pour la route de base et qu'on attend une réponse. Voici la commande pour lancer ce test :

```
php phpunit.phar
PHPUnit 4.8.21 by Sebastian Bergmann and contributors.
.Time: 3290ms, Memory: 21.50Mb  OK (1 test, 2 assertions)
```

On voit qu'ont été effectués un test et deux assertions et que tout s'est bien passé.

Environnement de test

Les tests s'effectuent dans un environnement particulier, ce qui est bien pratique. Regardez le fichier `phpunit.xml` :

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit backupGlobals="false"
         backupStaticAttributes="false"
         bootstrap="bootstrap/autoload.php"
         colors="true"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         processIsolation="false"
         stopOnFailure="false">
  <testsuites>
    <testsuite name="Application Test Suite">

      <directory suffix="Test.php">./tests</directory>
    </testsuite>
  </testsuites>
  <filter>
    <whitelist processUncoveredFilesFromWhitelist="true">
      <directory suffix=".php">./app</directory>
       <exclude>
           <file>./app/Http/routes.php</file>
       </exclude>
    </whitelist>
  </filter>
  <php>
    <env name="APP_ENV" value="testing"/>
    <env name="CACHE_DRIVER" value="array"/>
    <env name="SESSION_DRIVER" value="array"/>
    <env name="QUEUE_DRIVER" value="sync"/>
  </php>
</phpunit>
```

On trouve déjà quatre variables d'environnement :

- `APP_ENV` précise qu'on est en mode `testing` ;
- `CACHE_DRIVER` est en mode `array`, ce qui signifie que rien ne sera mis en cache pendant les tests (par défaut, on a `file`) ;

Copyright © 2016 Eyrolles.

- SESSION_DRIVER est en mode array, ce qui signifie qu'on ne va pas faire persister la session (par défaut, on a file) ;
- QUEUE_DRIVER : je n'évoque pas ici les tâches nécessaires ; on ne tiendra donc pas compte de cette variable.

On peut évidemment ajouter les variables dont on a besoin, par exemple si pendant les tests je ne veux plus MySQL mais SQLite. Il y a une variable à cet effet dans config/database.php :

```
<?php
'default'=>env('DB_CONNECTION', 'mysql'),
```

Il faut donc la renseigner dans .env :

```
DB_CONNECTION=mysql
```

Dans phpunit.xml, je peux écrire :

```
<env name="DB_CONNECTION" value="sqlite"/>
```

Pour les tests, je vais désormais utiliser sqlite.

Construire un test en trois étapes

Pour construire un test, on procède généralement en trois étapes :

1. on initialise les données ;
2. on agit sur ces données ;
3. on vérifie que le résultat est conforme à notre attente.

Comme tout cela est un peu abstrait, prenons un exemple. Remplacez le code de la méthode testBasicExample par le suivant, qui contient les trois étapes :

```
<?php
public function testBasicExample()
{
  $data=[10,20,30];
  $result=array_sum($data);
  $this->assertEquals(60, $result);
}
```

La méthode assertEquals sert à comparer deux valeurs, ici 60 et $result. Si vous lancez le test, vous obtenez ce qui suit :

```
OK (1 test, 1 assertion)
```

Copyright © 2016 Eyrolles.

Vous voyez à nouveau l'exécution d'un test et d'une assertion. Le tout s'est bien passé. Changez la valeur `60` par une autre et vous obtiendrez un échec :

```
FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Vous connaissez maintenant le principe de base d'un test et ce qu'on peut obtenir comme renseignement en cas d'échec.

Assertions et appel de routes

Assertions

Les assertions constituent l'outil de base des tests. On en a présenté une dans la section précédente, mais il en existe bien d'autres. Vous en trouvez la liste complète à l'adresse suivante : *http://phpunit.de/manual/current/en/appendixes.assertions.html*.

Voici un exemple d'utilisation de quelques assertions et d'un *helper* qu'on teste au passage :

```
<?php
public function testBasicExample()
{
  $data='Je suis petit';
  $this->assertTrue(starts_with($data, 'Je'));
  $this->assertFalse(starts_with($data, 'Tu'));
  $this->assertSame(starts_with($data, 'Tu'), false);
  $this->assertStringStartsWith('Je', $data);
  $this->assertStringEndsWith('petit', $data);
}
```

Lorsqu'on lance le test, on obtient un test et cinq assertions correctes :

```
OK (1 test, 5 assertions)
```

Appel de routes et test de réponse

Il est facile d'appeler une route pour effectuer un test sur la réponse. Modifiez la route de base pour celle-ci :

```
<?php
Route::get('/', function()
{
  return 'coucou';
});
```

Copyright © 2016 Eyrolles.

On a une requête avec l'URL de base et, comme réponse, la chaîne `'coucou'`. Nous allons tester la requête pour vérifier qu'elle aboutit bien, que la réponse est correcte et qu'elle vaut `'coucou'` :

```
<?php
public function testBasicExample()
{
  $response=$this->call('GET', '/');
  $this->assertResponseOk();
  $this->assertEquals('coucou', $response->getContent());
}
```

L'assertion `assertResponseOk` nous assure que la réponse est correcte. Ce n'est pas une assertion de PHPUnit, mais une spécifique de Laravel. La méthode `getContent` permet de lire la réponse. On obtient :

```
OK (1 test, 2 assertions)
```

Les vues et les contrôleurs

Vues

Qu'en est-il si on retourne une vue ? Écrivez le code suivant pour la route :

```
<?php
Route::get('/', function()
{
  return view('welcome')->with('message', 'Vous y êtes !');
});
```

Ajoutez ce qui suit dans cette vue :

```
{{ $message }}
```

Voici le test :

```
<?php
public function testBasicExample()
{
  $response=$this->call('GET', '/');
  $view=$response->original;
  $this->assertEquals('Vous y êtes !', $view['message']);
}
```

Copyright © 2016 Eyrolles.

On envoie la requête et on récupère la réponse. Ensuite, la méthode `original` nous récupère la vue. On peut alors tester la valeur de la variable `$message` dans la vue.

Laravel propose une autre assertion pour effectuer la même chose :

```
<?php
public function testBasicExample()
{
  $response=$this->call('GET', '/');
  $this->assertViewHas('message');
  $this->assertViewHas('message', 'Vous y êtes !');
}
```

L'assertion `assertViewHas` vérifie qu'une vue reçoit bien une donnée. On peut aussi, grâce au second paramètre, vérifier sa valeur.

Contrôleurs

Créez le contrôleur suivant :

```
<?php
namespace App\Http\Controllers;

class WelcomeController extends Controller
{
  public function index()
  {
    return view('welcome');
  }
}
```

Il est facile de tester les actions d'un contrôleur. Créez une route pour mettre en œuvre le contrôleur précédent :

```
<?php
Route::get('welcome', 'WelcomeController@index');
```

Vérifiez que cela fonctionne (vous aurez peut-être besoin de retoucher la vue où nous avons introduit une variable).

Supprimez le fichier `ExampleTest.php` qui ne va plus nous servir.

Créez l'architecture de dossiers suivante et le fichier `WelcomeControllerTest.php`.

Le dossier des tests

Copyright © 2016 Eyrolles.

Écrivez le code ci-après dans le fichier `WelcomeControllerTest.php` :

```
<?php
class WelcomeControllerTest extends TestCase
{
  public function testIndex()
  {
    $response=$this->action('GET', 'WelcomeController@index');
    $this->assertResponseOk();
  }
```

Une commande `Artisan` sert à créer une classe de test.

```
php artisan make:test WelcomeControllerTest
```

Si vous lancez le test, tout devrait bien se passer :

```
OK (1 test, 1 assertion)
```

Donc, pour appeler une action d'un contrôleur, il suffit d'utiliser la méthode `action` et de préciser le verbe, le nom du contrôleur et le nom de la méthode associée. De plus, il est important de bien organiser les tests pour s'y retrouver. Une bonne façon de procéder est d'adopter la même architecture de dossiers que celle prévue pour l'application.

Isoler les tests

Nous allons aborder un aspect important des tests qui ne s'appellent pas unitaires pour rien. Pour obtenir des tests efficaces, il faut bien les isoler, donc savoir ce qu'on teste, ne tester qu'une chose à la fois et ne pas mélanger les choses. C'est possible à condition que le code soit bien organisé.

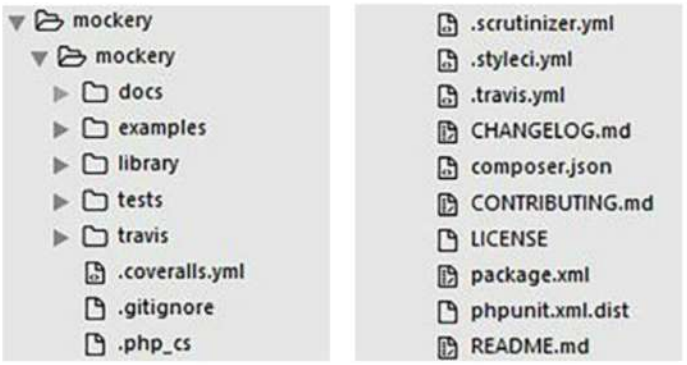
Avec `PHPUnit`, chaque test est effectué dans une application spécifique ; il n'est donc pas possible de les rendre dépendants les uns des autres.

En général, on utilise `Mockery` (*https://github.com/padraic/mockery*), un composant qui permet de simuler le comportement d'une classe. Il est déjà prévu dans l'installation de Laravel en mode développement :

```
"require-dev": {
  ...
  "mockery/mockery": "0.9.*"
},
```

Copyright © 2016 Eyrolles.

Le fait de prévoir ce composant uniquement pour le développement simplifie ensuite la mise en œuvre pour le déploiement. Normalement, vous devriez trouver ce composant dans vos dossiers.



Le composant Mockery

Simuler une classe

Mettons tout d'abord en place le code à tester. Ce ne sera pas trop réaliste, mais c'est juste pour comprendre le mécanisme de fonctionnement de `Mockery`. Remplacez le code du contrôleur `WelcomeController` par ce qui suit :

```
<?php
namespace App\Http\Controllers;

use Livre;

class WelcomeController extends Controller
{
  public function __construct()
  {
    $this->middleware('guest');
  }

  public function index(Livre $livre)
  {
    $livres=$livre->all();
    return view('welcome', compact('livres'));
  }
}
```

J'ai prévu l'injection d'un modèle dans la fonction `index`. Ce n'est pas forcément très judicieux d'un point de vue conceptuel et il vaudrait mieux passer par un repository comme nous l'avons vu plusieurs fois dans cet ouvrage, mais on va prendre cette situation simple pour exécuter un test. D'ailleurs, la classe `Livre` n'a pas besoin d'exister puisqu'on ne l'utilisera pas vraiment.

Copyright © 2016 Eyrolles.

La difficulté ici réside dans l'injection d'une classe. Comme on veut isoler les tests, l'idéal serait de simuler cette classe. C'est justement ce que permet `Mockery`.

Voici la classe de test que nous allons utiliser :

```
<?php
use Illuminate\Database\Eloquent\Collection;

class WelcomeControllerTest extends TestCase
{
  public function testIndex()
  {
    // Création de la collection
    $collection=new Collection; 1
    $i=2;
    while ($i--) {
      $collection->add((object) ['titre'=>'Titre' . $i]);
    }

    // Création Mock
    $mock=Mockery::mock('Livre'); 2
    $mock->shouldReceive('all') 3
         ->once()
         ->andReturn($collection);

    // Création lien
    $this->app->instance('Livre', $mock); 4

    // Action
    $response=$this->call('GET', 'welcome'); 5

    // Assertions
    $this->assertResponseOk(); 6
    $this->assertViewHas('livres');
  }

  public function tearDown() 7
  {
    Mockery::close();
  }
}
```

Et voici le code à ajouter dans la vue pour la rendre plus réaliste :

```
@foreach ($livres as $livre)
  <p>Titre : {{$livre->titre}}</p>
@endforeach
```

Si je lance le test, j'obtiens le résultat suivant :

```
OK (1 test, 2 assertions)
```

Copyright © 2016 Eyrolles.

Voyons de plus près ce code. Au départ, on crée une collection de livres 1. On crée ensuite un objet `Mock` en lui demandant de simuler la classe `Livre` 2. Puis, on définit le comportement qu'on désire pour cet objet 3. On lui dit que s'il reçoit l'appel de la méthode `all` (`shouldReceive`) une fois (`once`) il doit retourner la collection `$collection` (`andReturn`). On informe ensuite le conteneur de Laravel de la liaison entre la classe `Messages` et notre objet `Mock` 4. C'est une façon de dire à Laravel : « chaque fois que tu auras besoin de la classe `Livre`, tu utiliseras plutôt l'objet `Mock` ». Puis, on réalise l'action, ici la requête 5. On prévoit deux assertions : une pour vérifier qu'on a une réponse correcte et la seconde pour vérifier qu'on a bien les livres dans la vue 6. Pour finir, on prévoit pour travailler proprement de supprimer l'objet `Mock` à l'issue du test, dans la méthode `tearDown` qui sert justement à effectuer des actions après un test 7.

Vous connaissez maintenant le principe de l'utilisation de `Mockery`. Les possibilités sont vastes avec ce composant.

Il n'y a pas vraiment de règle quant à la constitution des tests, quant à ce qu'il faut tester ou pas. L'important est de comprendre comment les faire et de juger ce qui est utile ou pas selon les circonstances. Une façon efficace d'apprendre à réaliser des tests tout en comprenant mieux Laravel est de regarder comment ces tests ont été conçus (*https://github.com/laravel/framework/tree/master/tests*).

Tester une application

Laravel va encore plus loin dans la convivialité en facilitant le test d'une application.

Partez d'un Laravel tout neuf et ajoutez l'authentification avec la commande suivante :

```
php artisan make:auth
```

Prévoyez une base de données pour que cela fonctionne et entrez l'utilisateur `Toto`, avec l'adresse `toto@chez.fr` et le mot de passe `password`. On crée un test pour vérifier si cet utilisateur peut bien s'authentifier.

Commencez par supprimer le fichier `ExampleTest.php`.

Créez un nouveau fichier `AuthControllerTest.php` avec Artisan :

```
php artisan make:test AuthControllerTest
```

Complétez le code comme suit :

```
<?php
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;
```

Copyright © 2016 Eyrolles.

```
class AuthControllerTest extends TestCase
{
  /**
   * A basic test example.
   *
   * @return void
   */
  public function testRegister()
  {
    $this->visit('/')
         ->click('Login')
         ->type('toto@chez.fr', 'email')
         ->type('password', 'password')
         ->press('Login')
         ->see('Dashboard');
  }
}
```

Cela se lit comme de la prose ! Si vous lancez le test, vous obtiendrez :

```
OK (2 tests, 5 assertions)
```

Je ne détaillerai pas ici toutes les possibilités ; reportez-vous à la documentation : *https://laravel.com/docs/5.2/testing#application-testing*.

En résumé

- Laravel utilise `PHPUnit` pour effectuer les test unitaires.
- En plus des méthodes de `PHPUnit`, on dispose d'*helpers* pour intégrer les tests dans une application réalisée avec Laravel.
- Le composant `Mockery` permet de simuler le comportement d'une classe et donc de bien isoler les tests.
- Laravel propose des commandes conviviales pour tester une application.

Copyright © 2016 Eyrolles.

Copyright © 2016 Eyro les.

24 Événements et autorisations

Pour terminer, j'évoquerai dans ce chapitre les points qui peuvent constituer des alternatives intéressantes au codage présenté jusqu'à maintenant.

Nous allons ainsi passer en revue les événements (*events*) et les autorisations (*authorizations*).

Pour travailler, je vous propose d'installer l'application d'exemple que j'ai déposée sur https://github.com/bestmomo/laravel5-example. Elle vous donne l'occasion de voir un cas réaliste par rapport aux applications partielles qu'on a utilisées jusque-là.

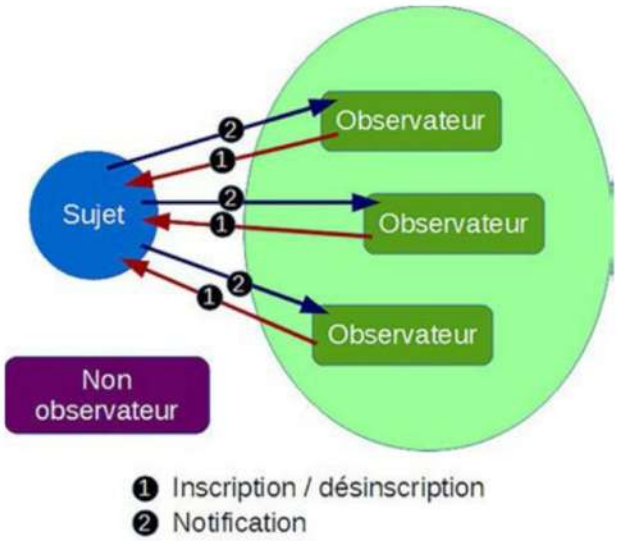
Explorez un peu l'application avant de poursuivre la lecture de ce chapitre.

Les événements

Le design pattern `Observateur` (`Observer`) établit une relation de type 1:n entre des objets. Si l'objet côté 1 change d'état, il en informe les objets côté n pour qu'ils puissent agir en conséquence. Cela implique une intendance et les objets doivent pouvoir s'inscrire et se mettre à l'écoute des événements du sujet.

Voici une visualisation de ce design pattern.

Copyright © 2016 Eyrolles.



Le design pattern Observateur

On a un sujet et des observateurs. Les observateurs doivent s'inscrire (on dit aussi s'abonner) auprès du sujet. Lorsque le sujet change d'état, il en informe tous ses abonnés. Un observateur peut aussi se désinscrire.

Vous trouverez une description détaillée de ce design pattern à l'adresse *http://design-patterns.fr/observateur*. Laravel l'implémente et le rend très simple d'utilisation, comme nous allons le voir.

Événements du framework

On peut utiliser les événements déjà présents dans le framework, qui en propose plusieurs au niveau de l'authentification :

- tentative de connexion (`login`) ;
- connexion (`login`) réussie ;
- déconnexion (`logout`) réussie.

On trouve aussi des événements avec `Eloquent` :

- `creating` ;
- `created` ;
- `updating` ;
- `updated` ;
- `saving` ;
- `saved` ;
- `deleting` ;
- `deleted` ;
- `restoring` ;
- `restored`.

Copyright © 2016 Eyrolles.

Fournisseur

Un fournisseur de services est dédié aux événements.

Le fournisseur de services pour les événements

En voici le code :

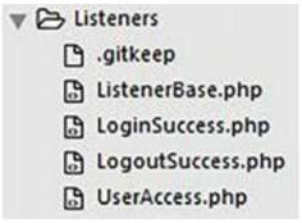
```
<?php
namespace App\Providers;

use Illuminate\Contracts\Events\Dispatcher as DispatcherContract;
use Illuminate\Foundation\Support\Providers\EventServiceProvider
as ServiceProvider;

class EventServiceProvider extends ServiceProvider {
  /**
   * The event handler mappings for the application.
   *
   * @var array
   */
  protected $listen=[ 1
    'Illuminate\Auth\Events\Login'=>['App\Listeners\LoginSuccess'],
    'Illuminate\Auth\Events\Logout'=> ['App\Listeners\LogoutSuccess'],
    'App\Event\UserAccess'=>['App\Listeners\UserAccess']
  ];

  /**
   * Register any other events for your application.
   *
   * @param \Illuminate\Contracts\Events\Dispatcher $events
   * @return void
   */
  public function boot(DispatcherContract $events)
  {
    parent::boot($events);
  }
}
```

On a une propriété `listen` pour les abonnements. C'est un simple tableau qui prend l'événement (`event`) comme clé et l'observateur (`listener`) comme valeur.

Dans l'application d'exemple, on veut savoir quand quelqu'un se connecte ou se déconnecte. On prévoit donc des observateurs dans le fournisseur 1.

Laravel prévoit de placer les observateurs (`listeners`) dans un dossier spécial en tant que classes.

Copyright © 2016 Eyrolles.



Le dossier des observateurs

Pour simplifier la gestion, le statut de l'utilisateur (son rôle) est enregistré dans la session. On a trois rôles :

- Administrateur (`admin`) : avec tous les droits ;
- Rédacteur (`redac`) : avec le droit de gérer des articles ;
- Utilisateur (`user`) : avec juste le droit de laisser des commentaires.

Dès que quelqu'un se connecte, on regarde son rôle et on le mémorise dans la session. Voici le code de l'observateur (`listener`) :

```
<?php
namespace App\Listeners;

use Illuminate\Auth\Events\Login;

class LoginSuccess extends ListenerBase
{
  /**
   * Handle the event.
   *
   * @param Login $login
   * @return void
   */
  public function handle(Login $login)
  {
    $this->status->setLoginStatus($login);
  }
}
```

Une propriété `status` apparaît. Pour en trouver l'origine, il faut voir que notre classe hérite de `ListenerBase` que voici :

Copyright © 2016 Eyrolles.

```
<?php
namespace App\Listeners;

use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;
use App\Services\Status;

class ListenerBase
{
  /**
   * The Status instance.
   *
   * @var App\Services\Status
   */
  protected $status;

  /**
   * Create the event listener.
   *
   * @param App\Services\Status $status
   * @return void
   */
  public function __construct(Status $status)
  {
    $this->status=$status;
  }
}
```

Cette classe de base est prévue pour justement factoriser l'affectation de la propriété. Si on regarde la méthode finalement visée, on voit qu'on mémorise l'information en session :

```
<?php
public function setLoginStatus($login)
{
  session()->put('status', $login->user->role->slug);
}
Et évidemment, en cas de déconnexion, c'est le même processus avec
l'observateur LogoutSuccess. Au final, on rend l'utilisateur simple
visiteur :
<?php
public function setVisitorStatus()
{
  session()->put('status', 'visitor');
}
```

Il faut considérer un dernier cas : lorsque l'utilisateur accède à une page, on va vérifier s'il y a un statut mémorisé en session et si ce n'est pas le cas en créer un.

Cette fois, ce n'est pas un événement qui existe comme `login` ou `logout` ; il nous faut donc le créer.

Laravel prévoit de placer les événements *(events)* dans un dossier spécial en tant que classes.

Copyright © 2016 Eyrolles.

Le dossier des événements

Dans le fichier `app/Http/Middlewares/App.php`, regardez le code suivant :

```
<?php
public function handle($request, Closure $next)
{
  ...
  event(new UserAccess);
  ...
}
```

J'utilise l'*helper* `event` pour déclencher un événement. Sans lui, il faudrait utiliser la façade correspondante :

```
<?php
Event::fire(new UserAccess);
```

Au niveau de l'inscription, j'ai donc ce qui suit :

```
<?php
protected $listen=[
  ...
  'App\Event\UserAccess'=>['App\Listeners\UserAccess']
];
```

Voici la classe de l'événement :

```
<?php
namespace App\Events;

use App\Events\Event;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class UserAccess extends Event
{
  use SerializesModels;

  /**
   * Get the channels the event should be broadcast on.
   *
   * @return array
   */
  public function broadcastOn()
```

Copyright © 2016 Eyrolles.

```
  {
    return [];
  }
}
```

Et voici l'observateur :

```
<?php
namespace App\Listeners;

use App\Events\UserAccess;

class UserAccess extends ListenerBase
{
  /**
   * Handle the event.
   *
   * @param UserAccess $event
   * @return void
   */
  public function handle(UserAccess $event)
  {
    $this->status->setStatus();
  }
}
```

On appelle donc la méthode dans le service :

```
<?php
public function setStatus()
{
  if (!session()->has('status'))
  {
    session()->put('status', auth()->check() ? auth()->user()->role-
>slug:'visitor');
  }
}
```

Créer un observateur

Dans l'application, tout est déjà en place. Comment fait-on pour créer un observateur ? Prenons le cas de la connexion.

Créons le squelette de l'observateur avec `Artisan` :

```
php artisan make:listener LoginSuccess --event=Illuminate\Auth\Events\Login
```

On donne le nom de l'observateur et celui de l'événement qu'on veut observer.

Copyright © 2016 Eyrolles.

Voici le code de l'observateur vierge :

```
<?php
namespace App\Listeners;

use Illuminate\Auth\Events\Login;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class LoginSuccess
{
  /**
   * Create the event listener.
   *
   * @return void
   */
  public function __construct()
  {
    //
  }

  /**
   *Handle the event.
   *
   * @param Login $event
   * @return void
   */
  public function handle(Login $event)
  {
    //
  }
}
```

Il suffit alors d'ajouter la logique de l'observateur dans la méthode `handle` de l'observateur (ou appeler la méthode d'une autre classe) :

```
<?php
public function handle(Login $login)
{
  $this->status->setLoginStatus($login);
}
```

Comme j'ai choisi d'utiliser une classe parente pour éviter de dupliquer du code, la propriété est créée dans celle-ci.

Créer un événement

Prenons l'exemple de `UserAccess`. Avec `Artisan`, je peux créer la classe :

```
php artisan make:event UserAccess
```


Copyright © 2016 Eyrolles.

En voici le code de base :

```
<?php
namespace App\Events;

use App\Events\Event;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class UserAccess extends Event
{
  use SerializesModels;

  /**
   * Create a new event instance.
   *
   * @return void
   */
  public function __construct()
  {
    //
  }

  /**
   * Get the channels the event should be broadcast on.
   *
   * @return array
   */
  public function broadcastOn()
  {
    return [];
  }
}
```

Il suffit ensuite de compléter selon les besoins. Dans le cas de l'application, il n'y a rien de spécial à effectuer.

Vous trouverez la documentation complète à l'adresse *https://laravel.com/docs/5.2/events*.

Les autorisations

Sécurité

Lorsqu'on développe une application, on prend de nombreuses précautions. Par exemple, les utilisateurs doivent s'authentifier pour éviter des actions non autorisées. Dans le code, on peut vérifier si la personne est authentifiée et quel est son degré d'habilitation. Dans l'application d'exemple, on met en session ce degré d'habilitation ; il suffit ensuite de le vérifier, par exemple dans les vues, pour adapter ce qui est affiché :

Copyright © 2016 Eyrolles.

```
@if (session('status')=='admin')
  {!! link_to_route('admin', trans('back/admin.administration'), [],
['class'=>'navbar-brand']) !!}
@else
  {!! link_to_route('blog.index', trans('back/admin.redaction'), [],
['class'=>'navbar-brand']) !!}
@endif
```

On peut aussi mettre en place des *middlewares* pour filtrer les accès. Dans l'application d'exemple, on a `isAdmin` :

```
<?php
namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\RedirectResponse;

class IsAdmin {
  /**
   * Handle an incoming request.
   *
   * @param \Illuminate\Http\Request $request
   * @param \Closure $next
   * @return mixed
   */
  public function handle($request, Closure $next)
  {
    if (session('status')==='admin')
    {
      return $next($request);
    }
    return new RedirectResponse(url('/'));
  }
}
```

Il suffit ensuite de l'utiliser dans les routes :

```
<?php
Route::get('admin', [
  'uses'=>'AdminController@admin',
  'as'=>'admin',
  'middleware'=>'admin'
]);
```

On peut aussi sécuriser l'application au niveau des requêtes de formulaires. Dans l'application d'exemple, la méthode `authorize` est située uniquement dans la classe `Request` qui est la base de toutes les autres :

Copyright © 2016 Eyrolles.

```
<?php
namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

abstract class Request extends FormRequest
{
  public function authorize()
  {
    // Honeypot
    return $this->input('address')=='';
  }
}
```

En effet, dans tous les formulaires il est prévu un contrôle caché. Par exemple, dans le formulaire de contact, on trouve le code suivant :

```
{!! Form::text('address', '', ['class'=>'hpet']) !!}
```

La classe `hpet` se contente de rendre le contrôle invisible. Un humain ne voit pas ce contrôle et donc ne le complète pas ; en revanche, un robot risque de l'utiliser. C'est ce qu'on appelle un pot de miel (pour attirer les abeilles virtuelles). On vérifie donc si ce contrôle est rempli et, si c'est le cas, on bloque le processus.

Différentes autorisations

En plus de ces possibilités, Laravel nous offre un système complet d'autorisations. Le plus simple pour voir comment cela fonctionne est de prendre un exemple. On a le dossier `app/Policies` avec un fichier.

Le dossier des autorisations

Voici le code de ce fichier :

```
<?php
namespace App\Policies;

use App\Models\Post;
use App\Models\User;

class PostPolicy
{
```

Copyright © 2016 Eyrolles.

```
    /**
     * Grant all abilities to administrator.
     *
     * @param \App\Models\User $user
     * @param string $ability
     * @return bool
     */
    public function before(User $user, $ability)
    {
        if (session('status')==='admin') {
            return true;
        }
    }

    /**
     * Determine if the given post can be changed by the user.
     *
     * @param \App\Models\User $user
     * @param \App\Models\Post $post
     * @return bool
     */
    public function change(User $user, Post $post)
    {
        return $user->id===$post->user_id;
    }
}
```

La méthode `before` est la première à être appelée. À ce niveau, on vérifie si l'utilisateur est un administrateur et donc possède tous les droits ; si c'est le cas, on renvoie `true`.

On a ensuite la méthode `change` avec comme paramètres l'utilisateur et l'article. Si l'utilisateur est le créateur de l'article on renvoie `true`.

Ces autorisations étant en place, il faut les déclarer. Regardez le fichier `app/Providers/AuthServiceProvider` :

```
<?php
namespace App\Providers;

use Illuminate\Contracts\Auth\Access\Gate as GateContract;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider
as ServiceProvider;
use App\Models\Post;
use App\Policies\PostPolicy;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies=[
        Post::class=>PostPolicy::class,
    ];
```

Copyright © 2016 Eyrolles.

```
    /**
     * Register any application authentication/authorization services.
     *
     * @param \Illuminate\Contracts\Auth\Access\Gate $gate
     * @return void
     */
    public function boot(GateContract $gate)
    {
        parent::registerPolicies($gate);
    }
}
```

Dans la propriété `policies`, on a prévu d'ajouter la classe `PostPolicy`. Il ne nous reste plus qu'à voir comment on l'utilise.

Par défaut, le contrôleur de base utilise le `trait AuthorizesRequests` ; donc, tous les contrôleurs connaissent les autorisations enregistrées.

Voici deux méthodes du contrôleur `BlogController` :

```
<?php
/**
 * Show the form for editing the specified resource.
 *
 * @param App\Repositories\UserRepository $user_gestion
 * @param int $id
 * @return Response
 */
public function edit(
    UserRepository $user_gestion,
    $id)
{
    $post=$this->blog_gestion->getByIdWithTags($id);
    $this->authorize('change', $post);
    $url=config('medias.url');
    return view('back.blog.edit', array_merge($this->blog_gestion-
>edit($post), compact('url')));
}

/**
 * Update the specified resource in storage.
 *
 * @param App\Http\Requests\PostUpdateRequest $request
 * @param int $id
 * @return Response
 */
public function update(
    PostRequest $request,
    $id)
{
    $post=$this->blog_gestion->getById($id);
    $this->authorize('change', $post);
    $this->blog_gestion->update($request->all(), $post);
    return redirect('blog')->with('ok', trans('back/blog.updated'));
}
```

Copyright © 2016 Eyrolles.

La première est destinée à afficher le formulaire de modification d'un article et la seconde à traiter la soumission des modifications. Dans les deux cas, on met en œuvre l'autorisation :

```
<?php
$this->authorize('change', $post);
```

Donc, s'il ne s'agit pas d'un administrateur ou du propriétaire de l'article, le processus se bloquera ici.

Vous trouverez la documentation complète à l'adresse *https://laravel.com/docs/5.2/authorization*.

En résumé

- Laravel comporte un système complet et simple de gestion des événements.
- Laravel intègre un système complet et simple de gestion des autorisations.

Copyright © 2016 Eyrolles.

Index

A

Ajax 285
authentification
  AuthController 139
  connexion 147
  déconnexion 152
  oubli du mot de passe 153
  PasswordController 141
  présentation 133
autorisation 319

C

commande 205
composer 9
configuration 65
contrôleur 39, 49, 62, 69, 92, 103, 170, 293
courriel 63

D

déploiement 243

E

Eloquent 89, 125, 168, 189, 275
entrée 45
environnement 17, 241
erreur 128
événement 311

F

façade 27, 264
formulaire 47, 55, 71, 93, 120, 152, 181, 203, 256
fournisseur de services 29, 80, 252, 264, 313

H

htaccess 14

I

injection de dépendances 75
installation 9

L

laravelcollective 20
localisation 269

M

macro 249
middleware
  admin 173
  ajax 295

Copyright © 2016 Eyrolles.

Authenticate 136
gestion de la locale 272
présentation 136
RedirectIfAuthenticated 137
migration
création 87
exemple 161, 184, 223
installation 86
population 164, 187
présentation 85
utilisation 88

P

protection CSRF 51

Q

Query Builder 223

R

redirection 38
relation
1\n 161
relations
n\n 183
réponse 31
requête de formulaire 59, 68, 90, 107, 174, 190
ressource 101, 103, 111
routage
contrainte 26
nom 27
paramètres 24
présentation 21

S

session 67, 272, 314

T

test unitaire 297

V

validation 53
vue 32, 55, 71, 93, 117, 177, 199, 249, 259, 285